Mixing Metaphors: Actors as Channels and Channels as Actors

Mixing Metaphors: Actors as Channels and Channels as Actors*

Simon Fowler[1], Sam Lindley[2], and Philip Wadler[3]

1 **University of Edinburgh, Edinburgh, United Kingdom**
`simon.fowler@ed.ac.uk`
2 **University of Edinburgh, Edinburgh, United Kingdom**
`sam.lindley@ed.ac.uk`
3 **University of Edinburgh, Edinburgh, United Kingdom**
`wadler@inf.ed.ac.uk`

Abstract

Channel- and actor-based programming languages are both used in practice, but the two are often confused. Languages such as Go provide anonymous processes which communicate using buffers or rendezvous points—known as channels—while languages such as Erlang provide addressable processes—known as actors—each with a single incoming message queue. The lack of a common representation makes it difficult to reason about translations that exist in the folklore. We define a calculus λ_{ch} for typed asynchronous channels, and a calculus λ_{act} for typed actors. We define translations from λ_{act} into λ_{ch} and λ_{ch} into λ_{act} and prove that both are type- and semantics-preserving. We show that our approach accounts for synchronisation and selective receive in actor systems and discuss future extensions to support guarded choice and behavioural types.

1998 ACM Subject Classification D.1.3 Concurrent Programming

Keywords and phrases Actors, Channels, Communication-centric Programming Languages

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.11

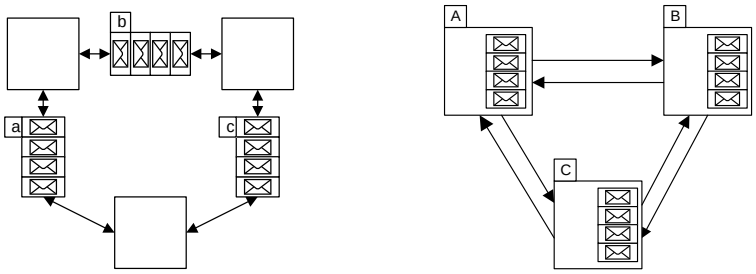
1 Introduction

When comparing channels (as used by Go) and actors (as used by Erlang), one runs into an immediate mixing of metaphors. The words themselves do not refer to comparable entities!

In languages such as Go, anonymous processes pass messages via named channels, whereas in languages such as Erlang, named processes accept messages from an associated mailbox. A channel is either a named rendezvous point or buffer, whereas an actor is a process. We should really be comparing named processes (actors) with anonymous processes, and buffers tied to a particular process (mailboxes) with buffers that can link any process to any process (channels). Nonetheless, we will stick with the popular names, even if it is as inapposite as comparing TV channels with TV actors.

Figure 1 compares asynchronous channels with actors. On the left, three anonymous processes communicate via channels named a, b, c. On the right, three processes named A, B, C send messages to each others' associated mailboxes. Actors are necessarily asynchronous, allowing non-blocking sends and buffering of received values, whereas channels can either be asynchronous or synchronous (rendezvous-based). Indeed, Go provides both synchronous *and* asynchronous channels, and libraries such as `core.async` [24] provide library support

* This work was supported by EPSRC grants EP/L01503X/1 (University of Edinburgh CDT in Pervasive Parallelism) and EP/K034413/1 (A Basis for Concurrency and Distribution).



(a) Asynchronous Channels. **(b)** Actors.

Figure 1 Channels and Actors.

for asynchronous channels. However, this is not the only difference: each actor has a single buffer which only it can read—its *mailbox*—whereas asynchronous channels are free-floating buffers that can be read by any process with a reference to the channel.

Channel-based languages such as Go enjoy a firm basis in process calculi such as CSP [25] and the π-calculus [38]. It is easy to type channels, either with simple types (see [46], p. 231) or more complex systems such as session types [26, 27, 17]. Actor-based languages such as Erlang are seen by many as the "gold standard" for distributed computing due to their support for fault tolerance through supervision hierarchies [6, 7].

Both models are popular with developers, with channel-based languages and frameworks such as Go, Concurrent ML [45], and Hopac [28]; and actor-based languages and frameworks such as Erlang, Elixir, and Akka.

1.1 Motivation

This paper provides a formal account of actors and channels as implemented in programming languages. Our motivation for a formal account is threefold: it helps clear up confusion; it clarifies results that have been described informally by putting practice into theory; and it provides a foundation for future research.

Confusion. There is often confusion over the differences between channels and actors. For example, the following questions appear on StackOverflow and Quora respectively:

> "If I wanted to port a Go library that uses Goroutines, would Scala be a good choice because its inbox/[A]kka framework is similar in nature to coroutines?" [31], and

> "I don't know anything about [the] actor pattern however I do know goroutines and channels in Go. How are [the] two related to each other?" [29]

In academic circles, the term *actor* is often used imprecisely. For instance, Albert et al. [5] refer to Go as an actor language. Similarly, Harvey [21] refers to his language Ensemble as actor-based. Ensemble is a language specialised for writing distributed applications running on heterogeneous platforms. It is actor-based to the extent that it has lightweight, addressable, single-threaded processes, and forbids co-ordination via shared memory. However, Ensemble communicates using channels as opposed to mailboxes so we would argue that it is channel-based (with actor-like features) rather than actor-based.

Putting practice into theory. The success of actor-based languages is largely due to their support for *supervision*. A popular pattern for writing actor-based applications is to arrange processes in *supervision hierarchies* [6], where *supervisor* processes restart child processes should they fail. Projects such as Proto.Actor [44] emulate actor-style programming in a channel-based language in an attempt to gain some of the benefits, by associating queues with processes. Hopac [28] is a channel-based library for F#, based on Concurrent ML [45]. The documentation [1] contains a comparison with actors, including an implementation of a simple actor-based communication model using Hopac-style channels, as well as an implementation of Hopac-style channels using an actor-based communication model. By comparing the two, this paper provides a formal model for the underlying techniques, and studies properties arising from the translations.

A foundation for future research. Traditionally, actor-based languages have had untyped mailboxes. More recent advancements such as TAkka [22], Akka Typed [4], and Typed Actors [47] have added types to mailboxes in order to gain additional safety guarantees. Our formal model provides a foundation for these innovations, characterises why naïvely adding types to mailboxes is problematic, and provides a core language for future experimentation.

1.2 Our approach

We define two concurrent λ-*calculi*, describing *asynchronous* channels and type-parameterised actors, define translations between them, and then discuss various extensions.

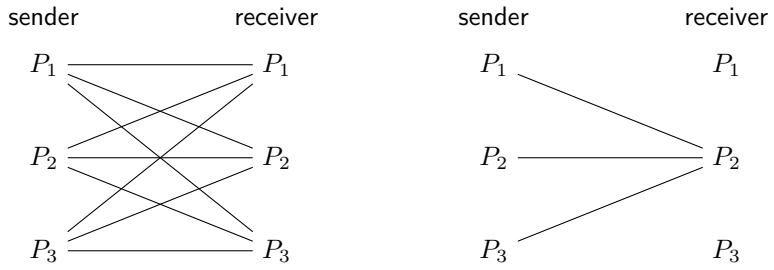
Why the λ calculus? Our common framework is that of a simply-typed concurrent λ-calculus: that is, a λ-calculus equipping a term language with primitives for communication and concurrency, as well as a language of *configurations* to model concurrent behaviour. We work with the λ-calculus rather than a process calculus for two reasons: firstly, the simply-typed λ-calculus has a well-behaved core with a strong metatheory (for example, confluent reduction and strong normalisation), as well as a direct propositions-as-types correspondence with logic. We can therefore modularly extend the language, knowing which properties remain; typed process calculi typically do not have such a well-behaved core.

Secondly, we are ultimately interested in functional programming languages; the λ calculus is the canonical choice for studying such extensions.

Why asynchronous channels? While actor-based languages must be asynchronous by design, channels may be either synchronous (requiring a rendezvous between sender and receiver) or asynchronous (where sending happens immediately). In this paper, we consider asynchronous channels since actors must be asynchronous, and it is possible to emulate asynchronous channels using synchronous channels [45]. We could adopt synchronous channels, use these to encode asynchronous channels, and then do the translations. We elect not to since it complicates the translations, and we argue that the distinction between synchronous and asynchronous communication is not *the* defining difference between the two models.

1.3 Summary of results

We identify four key differences between the models, which are exemplified by the formalisms and the translations: process addressability, the restrictiveness of communication patterns, the granularity of typing, and the ability to control the order in which messages are processed.



(a) Channel. **(b)** Mailbox.

Figure 2 Mailboxes as pinned channels.

Process addressability. In channel-based systems, processes are *anonymous*, whereas channels are named. In contrast, in actor-based systems, processes are named.

Restrictiveness of communication patterns. Communication over full-duplex channels is more liberal than communication via mailboxes, as shown in Figure 2. Figure 2a shows the communication patterns allowed by a single channel: each process P_i can use the channel to communicate with every other process. Conversely, Figure 2b shows the communication patterns allowed by a mailbox associated with process P_2: while any process can send to the mailbox, only P_2 can read from it. Viewed this way, it is apparent that the restrictions imposed on the communication behaviour of actors are exactly those captured by Merro and Sangiorgi's localised π-calculus [37].

Readers familiar with actor-based programming may be wondering whether such a characterisation is too crude, as it does not account for processing messages out-of-order. Fear not—we show in Section 7 that our minimal actor calculus can simulate this functionality.

Restrictiveness of communication patterns is not necessarily a bad thing; while it is easy to distribute actors, *delegation* of asynchronous channels is more involved, requiring a distributed algorithm [30]. Associating mailboxes with addressable processes also helps with structuring applications for reliability [7].

Granularity of typing. As a result of the fact that each process has a single incoming message queue, mailbox types tend to be less precise; in particular, they are most commonly variant types detailing all of the messages that can be received. Naïvely implemented, this gives rise to the *type pollution problem*, which we describe further in Section 2.

Message ordering. Channels and mailboxes are ordered message queues, but there is no inherent ordering between messages on two different channels. Channel-based languages allow a user to specify from which channel a message should be received, whereas processing messages out-of-order can be achieved in actor languages using selective receive.

The remainder of the paper captures these differences both in the design of the formalisms, and the techniques used in the encodings and extensions.

1.4 Contributions and paper outline

This paper makes five main contributions:

1. A calculus λ_{ch} with typed asynchronous channels (Section 3), and a calculus λ_{act} with type-parameterised actors (Section 4), based on the λ-calculus extended with communication

```
chanStack(ch) ≜ rec loop(st).
    let cmd ⇐ take ch in
    case cmd {
      Push(v) ↦ loop(v :: st)
      Pop(resCh) ↦
        case st {
          [ ] ↦ give (None) resCh;
                loop [ ]
          x :: xs ↦ give (Some(x)) resCh;
                loop xs   }
    }

chanClient(stackCh) ≜
  give (Push(5)) stackCh;
  let resCh ⇐ newCh in
  give (Pop(resCh)) stackCh;
  take resCh

chanMain ≜
  let stackCh ⇐ newCh in
  fork (chanStack(stackCh) [ ]);
  chanClient(stackCh)
```

(a) Channel-based stack.

```
actorStack ≜ rec loop(st).
    let cmd ⇐ receive in
    case cmd {
      Push(v) ↦ loop(v :: st)
      Pop(resPid) ↦
        case st {
          [ ] ↦ send (None) resPid;
                loop [ ]
          x :: xs ↦ send (Some(x)) resPid;
                loop xs   }
    }

actorClient(stackPid) ≜
  send (Push(5)) stackPid;
  let selfPid ⇐ self in
  send (Pop(selfPid)) stackPid;
  receive

actorMain ≜
  let stackPid ⇐ spawn (actorStack [ ]) in
  actorClient(stackPid)
```

(b) Actor-based stack.

Figure 3 Concurrent stacks using channels and actors.

primitives specialised to each model. We give a type system and operational semantics for each calculus, and precisely characterise the notion of progress that each calculus enjoys.

2. A simple translation from λ_{act} into λ_{ch} (Section 5), and a more involved translation from λ_{ch} into λ_{act} (Section 6), with proofs that both translations are type- and semantics-preserving. While the former translation is straightforward, it is *global*, in the sense of Felleisen [12]. While the latter is more involved, it is in fact *local*. Our initial translation from λ_{ch} to λ_{act} sidesteps type pollution by assigning the same type to each channel in the system.
3. An extension of λ_{act} to support synchronous calls, showing how this can alleviate type pollution and simplify the translation from λ_{ch} into λ_{act} (Section 7.1).
4. An extension of λ_{act} to support Erlang-style selective receive, a translation from λ_{act} with selective receive into plain λ_{act}, and proofs that the translation is type- and semantics-preserving (Section 7.2).
5. An extension of λ_{ch} with input-guarded choice (Section 7.3) and an outline of how λ_{act} might be extended with behavioural types (Section 7.4).

The rest of the paper is organised as follows: Section 2 displays side-by-side two implementations of a concurrent stack, one using channels and the other using actors; Section 3–7 presents the main technical content; Section 8 discusses related work; and Section 9 concludes.

2 Channels and actors side-by-side

Let us consider the example of a concurrent stack. A concurrent stack carrying values of type A can receive a command to push a value onto the top of the stack, or to pop a value and return

```
chanClient2(intStackCh, stringStackCh) ≜
  let intResCh ⇐ newCh in
  let strResCh ⇐ newCh in
  give (Pop(intResCh)) intStackCh;
  let res1 ⇐ take intResCh in
  give (Pop(strResCh)) stringStackCh;
  let res2 ⇐ take strResCh in
  (res1, res2)
```

```
actorClient2(intStackPid, stringStackPid) ≜
  let selfPid ⇐ self in
  send (Pop(selfPid)) intStackPid;
  let res1 ⇐ receive in
  send (Pop(selfPid)) stringStackPid;
  let res2 ⇐ receive in
  (res1, res2)
```

Figure 4 Clients interacting with multiple stacks.

it to the process making the request. Assuming a standard encoding of algebraic datatypes, we define a type Operation(A) = Push(A) | Pop(B) (where B = ChanRef(A) for channels, and ActorRef(A) for actors) to describe operations on the stack, and Option(A) = Some(A) | None to handle the possibility of popping from an empty stack.

Figure 3 shows the stack implemented using channels (Figure 3a) and using actors (Figure 3b). Each implementation uses a common core language based on the simply-typed λ-calculus extended with recursion, lists, and sums.

At first glance, the two stack implementations seem remarkably similar. Each:

1. Waits for a command
2. Case splits on the command, and either:
   - Pushes a value onto the top of the stack, or;
   - Takes the value from the head of the stack and returns it in a response message
3. Loops with an updated state.

The main difference is that **chanStack** is parameterised over a channel *ch*, and retrieves a value from the channel using take *ch*. Conversely, **actorStack** retrieves a value from its mailbox using the nullary primitive receive.

Let us now consider functions which interact with the stacks. The **chanClient** function sends commands over the *stackCh* channel, and begins by pushing 5 onto the stack. Next, it creates a channel *resCh* to be used to receive the result and sends this in a request, before retrieving the result from the result channel using take. In contrast, **actorClient** performs a similar set of steps, but sends its process ID (retrieved using self) in the request instead of creating a new channel; the result is then retrieved from the mailbox using receive.

Type pollution. The differences become more prominent when considering clients which interact with multiple stacks of different types, as shown in Figure 4. Here, **chanClient2** creates new result channels for integers and strings, sends requests for the results, and creates a pair of type (Option(Int) × Option(String)). The **actorClient2** function attempts to do something similar, but cannot create separate result channels. Consequently, the actor must be able to handle messages either of type Option(Int) *or* type Option(String), meaning that the final pair has type (Option(Int) + Option(String)) × (Option(Int) + Option(String)).

Additionally, it is necessary to modify **actorStack** to use the correct injection into the actor type when sending the result; for example an integer stack would have to send a value inl (Some(5)) instead of simply Some(5). This *type pollution* problem can be addressed through the use of subtyping [22], or synchronisation abstractions such as futures [10].

Syntax

$$
\begin{array}{lrl}
\text{Types} & A, B & ::= \mathbf{1} \mid A \to B \mid \mathsf{ChanRef}(A) \\
\text{Variables and names} & \alpha & ::= x \mid a \\
\text{Values} & V, W & ::= \alpha \mid \lambda x.M \mid () \\
\text{Computations} & L, M, N & ::= V\,W \\
 & & \mid \mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N \mid \mathsf{return}\ V \\
 & & \mid \mathsf{fork}\ M \mid \mathsf{give}\ V\ W \mid \mathsf{take}\ V \mid \mathsf{newCh}
\end{array}
$$

Value typing rules $\boxed{\Gamma \vdash V : A}$

$$\textsc{Var}\ \frac{\alpha : A \in \Gamma}{\Gamma \vdash \alpha : A} \qquad \textsc{Abs}\ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B} \qquad \textsc{Unit}\ \frac{}{\Gamma \vdash () : \mathbf{1}}$$

Computation typing rules $\boxed{\Gamma \vdash M : A}$

$$\textsc{App}\ \frac{\Gamma \vdash V : A \to B \quad \Gamma \vdash W : A}{\Gamma \vdash V\,W : B} \qquad \textsc{EffLet}\ \frac{\Gamma \vdash M : A \quad \Gamma, x : A \vdash N : B}{\Gamma \vdash \mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N : B} \qquad \textsc{Return}\ \frac{\Gamma \vdash V : A}{\Gamma \vdash \mathsf{return}\ V : A}$$

$$\textsc{Give}\ \frac{\Gamma \vdash V : A \quad \Gamma \vdash W : \mathsf{ChanRef}(A)}{\Gamma \vdash \mathsf{give}\ V\ W : \mathbf{1}} \qquad \textsc{Take}\ \frac{\Gamma \vdash V : \mathsf{ChanRef}(A)}{\Gamma \vdash \mathsf{take}\ V : A} \qquad \textsc{Fork}\ \frac{\Gamma \vdash M : \mathbf{1}}{\Gamma \vdash \mathsf{fork}\ M : \mathbf{1}} \qquad \textsc{NewCh}\ \frac{}{\Gamma \vdash \mathsf{newCh} : \mathsf{ChanRef}(A)}$$

Figure 5 Syntax and typing rules for λ_{ch} terms and values.

3 λ_{ch}: A concurrent λ-calculus for channels

In this section we introduce λ_{ch}, a concurrent λ-calculus extended with asynchronous channels. To concentrate on the core differences between channel- and actor-style communication, we begin with minimal calculi; note that these do not contain all features (such as lists, sums, and recursion) needed to express the examples in Section 2.

3.1 Syntax and typing of terms

Figure 5 gives the syntax and typing rules of λ_{ch}, a λ-calculus based on fine-grain call-by-value [34]: terms are partitioned into values and computations. Key to this formulation are two constructs: $\mathsf{return}\ V$ represents a computation that has completed, whereas $\mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N$ evaluates M to $\mathsf{return}\ V$, substituting V for x in M. Fine-grain call-by-value is convenient since it makes evaluation order explicit and, unlike A-normal form [13], is closed under reduction.

Types consist of the unit type $\mathbf{1}$, function types $A \to B$, and channel reference types $\mathsf{ChanRef}(A)$ which can be used to communicate along a channel of type A. We let α range over variables x and runtime names a. We write $\mathsf{let}\ x = V\ \mathsf{in}\ M$ for $(\lambda x.M)\,V$ and $M; N$ for $\mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N$, where x is fresh.

Communication and concurrency for channels. The $\mathsf{give}\,V\,W$ operation sends value V along channel W, while $\mathsf{take}\,V$ retrieves a value from a channel V. Assuming an extension of the language with integers and arithmetic operators, we can define a function $\mathtt{neg}(c)$ which receives a number n along channel c and replies with the negation of n as follows:

$$\mathtt{neg}(c) \triangleq \mathsf{let}\ n \Leftarrow \mathsf{take}\,c\ \mathsf{in}\ \mathsf{let}\ negN \Leftarrow (-n)\ \mathsf{in}\ \mathsf{give}\ negN\ c$$

Syntax of evaluation contexts and configurations

Evaluation contexts	$E ::= [\,] \mid \mathsf{let}\ x \Leftarrow E\ \mathsf{in}\ M$
Configurations	$\mathcal{C}, \mathcal{D}, \mathcal{E} ::= \mathcal{C} \parallel \mathcal{D} \mid (\nu a)\mathcal{C} \mid a(\overrightarrow{V}) \mid M$
Configuration contexts	$G ::= [\,] \mid G \parallel \mathcal{C} \mid (\nu a)G$

Typing rules for configurations $\boxed{\Gamma; \Delta \vdash \mathcal{C}}$

$$\textsc{Par}\ \frac{\Gamma; \Delta_1 \vdash \mathcal{C}_1 \quad \Gamma; \Delta_2 \vdash \mathcal{C}_2}{\Gamma; \Delta_1, \Delta_2 \vdash \mathcal{C}_1 \parallel \mathcal{C}_2} \qquad \textsc{Chan}\ \frac{\Gamma, a : \mathsf{ChanRef}(A); \Delta, a{:}A \vdash \mathcal{C}}{\Gamma; \Delta \vdash (\nu a)\mathcal{C}} \qquad \textsc{Buf}\ \frac{(\Gamma \vdash V_i : A)_i}{\Gamma; a : A \vdash a(\overrightarrow{V})} \qquad \textsc{Term}\ \frac{\Gamma \vdash M : \mathbf{1}}{\Gamma; \cdot \vdash M}$$

Figure 6 λ_{ch} configurations and evaluation contexts.

The fork M operation spawns a new process to evaluate term M. The operation returns the unit value, and therefore it is not possible to interact with the process directly. The newCh operation creates a new channel. Note that channel creation is decoupled from process creation, meaning that a process can have access to multiple channels.

3.2 Operational semantics

Configurations. The concurrent behaviour of λ_{ch} is given by a nondeterministic reduction relation on *configurations* (Figure 6). Configurations consist of parallel composition ($\mathcal{C} \parallel \mathcal{D}$), restrictions ($(\nu a)\mathcal{C}$), computations ($M$), and buffers ($a(\overrightarrow{V})$, where $\overrightarrow{V} = V_1 \cdot \ldots \cdot V_n$).

Evaluation contexts. Reduction is defined in terms of evaluation contexts E, which are simplified due to fine-grain call-by-value. We also define configuration contexts, allowing reduction modulo parallel composition and name restriction.

Reduction. Figure 7 shows the reduction rules for λ_{ch}. Reduction is defined as a deterministic reduction on terms ($\longrightarrow_{\mathsf{M}}$) and a nondeterministic reduction relation on configurations ($\longrightarrow$). Reduction on configurations is defined modulo structural congruence rules which capture scope extrusion and the commutativity and associativity of parallel composition.

Typing of configurations. To ensure that buffers are well-scoped and contain values of the correct type, we define typing rules on configurations (Figure 6). The judgement $\Gamma; \Delta \vdash \mathcal{C}$ states that under environments Γ and Δ, $\mathcal{C}$ is well-typed. Γ is a typing environment for terms, whereas Δ is a linear typing environment for configurations, mapping names a to channel types A. Linearity in Δ ensures that a configuration $\mathcal{C}$ under a name restriction $(\nu a)\mathcal{C}$ contains exactly one buffer with name a. Note that Chan extends both Γ *and* Δ, adding an (unrestricted) *reference* into Γ and the *capability* to type a buffer into Δ. Par states that $\mathcal{C}_1 \parallel \mathcal{C}_2$ is typeable if $\mathcal{C}_1$ and $\mathcal{C}_2$ are typeable under disjoint linear environments, and Buf states that under a term environment Γ and a singleton linear environment $a{:}A$, it is possible to type a buffer $a(\overrightarrow{V})$ if $\Gamma \vdash V_i{:}A$ for all $V_i \in \overrightarrow{V}$. As an example, $(\nu a)(a(\overrightarrow{V}))$ is well-typed, but $(\nu a)(a(\overrightarrow{V}) \parallel a(\overrightarrow{W}))$ and $(\nu a)(\mathsf{return}\ ())$ are not.

Relation notation. Given a relation R, we write R^+ for its transitive closure, and R^* for its reflexive, transitive closure.

Reduction on terms

$$(\lambda x.M)\,V \longrightarrow_{\mathsf{M}} M\{V/x\} \qquad \mathsf{let}\ x \Leftarrow \mathsf{return}\ V\ \mathsf{in}\ M \longrightarrow_{\mathsf{M}} M\{V/x\} \qquad E[M_1] \longrightarrow_{\mathsf{M}} E[M_2] \ (\text{if } M_1 \longrightarrow_{\mathsf{M}} M_2)$$

Structural congruence

$$\mathcal{C} \parallel \mathcal{D} \equiv \mathcal{D} \parallel C \qquad \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) \equiv (\mathcal{C} \parallel D) \parallel \mathcal{E} \qquad \mathcal{C} \parallel (\nu a)\mathcal{D} \equiv (\nu a)(\mathcal{C} \parallel \mathcal{D}) \text{ if } a \notin \mathsf{fv}(\mathcal{C})$$

$$G[\mathcal{C}] \equiv G[\mathcal{D}] \text{ if } \mathcal{C} \equiv \mathcal{D}$$

Reduction on configurations

$$\begin{array}{lrcll}
\textsc{Give} & E[\mathsf{give}\,W\,a] \parallel a(\overrightarrow{V}) & \longrightarrow & E[\mathsf{return}\,()] \parallel a(\overrightarrow{V} \cdot W) & \\
\textsc{Take} & E[\mathsf{take}\,a] \parallel a(W \cdot \overrightarrow{V}) & \longrightarrow & E[\mathsf{return}\,W] \parallel a(\overrightarrow{V}) & \\
\textsc{Fork} & E[\mathsf{fork}\,M] & \longrightarrow & E[\mathsf{return}\,()] \parallel M & \\
\textsc{NewCh} & E[\mathsf{newCh}] & \longrightarrow & (\nu a)(E[\mathsf{return}\,a] \parallel a(\epsilon)) & (a \text{ is a fresh name}) \\
\textsc{LiftM} & G[M_1] & \longrightarrow & G[M_2] & (\text{if } M_1 \longrightarrow_{\mathsf{M}} M_2) \\
\textsc{Lift} & G[\mathcal{C}_1] & \longrightarrow & G[\mathcal{C}_2] & (\text{if } \mathcal{C}_1 \longrightarrow \mathcal{C}_2)
\end{array}$$

Figure 7 Reduction on λ_{ch} terms and configurations.

Properties of the term language. Reduction on terms preserves typing, and pure terms enjoy progress. We omit most proofs in the body of the paper which are mainly straightforward inductions; selected full proofs can be found in the extended version [15].

▶ **Lemma 1** (Preservation (λ_{ch} terms)). *If $\Gamma \vdash M : A$ and $M \longrightarrow_{\mathsf{M}} M'$, then $\Gamma \vdash M' : A$.*

▶ **Lemma 2** (Progress (λ_{ch} terms)). *Assume Γ is empty or only contains channel references $a_i{:}\mathsf{ChanRef}(A_i)$. If $\Gamma \vdash M{:}A$, then either:*

1. *$M = \mathsf{return}\,V$ for some value V, or*
2. *M can be written $E[M']$, where M' is a communication or concurrency primitive (i.e., $\mathsf{give}\,V\,W$, $\mathsf{take}\,V$, $\mathsf{fork}\,M$, or newCh), or*
3. *There exists some M' such that $M \longrightarrow_{\mathsf{M}} M'$.*

Reduction on configurations. Concurrency and communication is captured by reduction on configurations. Reduction is defined modulo structural congruence rules, which capture the associativity and commutativity of parallel composition, as well as the usual scope extrusion rule. The Give rule reduces $\mathsf{give}\,W\,a$ in parallel with a buffer $a(\overrightarrow{V})$ by adding the value W onto the end of the buffer. The Take rule reduces $\mathsf{take}\,a$ in parallel with a non-empty buffer by returning the first value in the buffer. The Fork rule reduces $\mathsf{fork}\,M$ by spawning a new thread M in parallel with the parent process. The NewCh rule reduces newCh by creating an empty buffer and returning a fresh name for that buffer.

Structural congruence and reduction preserve the typeability of configurations.

▶ **Lemma 3.** *If $\Gamma; \Delta \vdash \mathcal{C}$ and $\mathcal{C} \equiv \mathcal{D}$ for some configuration $\mathcal{D}$, then $\Gamma; \Delta \vdash \mathcal{D}$.*

▶ **Theorem 4** (Preservation (λ_{ch} configurations)). *If $\Gamma; \Delta \vdash \mathcal{C}_1$ and $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$ then $\Gamma; \Delta \vdash \mathcal{C}_2$.*

3.3 Progress and canonical forms

While it is possible to prove deadlock-freedom in systems with more discerning type systems based on linear logic [48, 35] or those using channel priorities [41], more liberal calculi such

as λ_{ch} and λ_{act} allow deadlocked configurations. We thus define a form of progress which does not preclude deadlock; to help with proving a progress result, it is useful to consider the notion of a *canonical form* in order to allow us to reason about the configuration as a whole.

▶ **Definition 5** (Canonical form (λ_{ch})). A configuration $\mathcal{C}$ is in *canonical form* if it can be written $(\nu a_1) \dots (\nu a_n)(M_1 \parallel \dots \parallel M_m \parallel a_1(\overrightarrow{V_1}) \parallel \dots \parallel a_n(\overrightarrow{V_n}))$.

Well-typed open configurations can be written in a form similar to canonical form, but without bindings for names already in the environment. An immediate corollary is that well-typed closed configurations can always be written in a canonical form.

▶ **Lemma 6.** *If $\Gamma; \Delta \vdash \mathcal{C}$ with $\Delta = a_1 : A_1, \dots, a_k : A_k$, then there exists a $\mathcal{C}' \equiv \mathcal{C}$ such that $\mathcal{C}' = (\nu a_{k+1}) \dots (\nu a_n)(M_1 \parallel \dots \parallel M_m \parallel a_1(\overrightarrow{V_1}) \parallel \dots \parallel a_n(\overrightarrow{V_n}))$.*

▶ **Corollary 7.** *If $\cdot; \cdot \vdash \mathcal{C}$, then there exists some $\mathcal{C}' \equiv \mathcal{C}$ such that $\mathcal{C}'$ is in canonical form.*

Armed with a canonical form, we can now state that the only situation in which a well-typed closed configuration cannot reduce further is if all threads are either blocked or fully evaluated. Let a *leaf configuration* be a configuration without subconfigurations, i.e., a term or a buffer.

▶ **Theorem 8** (Weak progress (λ_{ch} configurations)).
Let $\cdot; \cdot \vdash \mathcal{C}$, $\mathcal{C} \not\longrightarrow$, and let $\mathcal{C}' = (\nu a_1) \dots (\nu a_n)(M_1 \parallel \dots \parallel M_m \parallel a_1(\overrightarrow{V_1}) \parallel \dots a_n(\overrightarrow{V_n}))$ be a canonical form of $\mathcal{C}$. Then every leaf of $\mathcal{C}$ is either:

1. *A buffer $a_i(\overrightarrow{V_i})$;*
2. *A fully-reduced term of the form $\mathsf{return}\, V$, or;*
3. *A term of the form $E[\mathsf{take}\, a_i]$, where $\overrightarrow{V_i} = \epsilon$.*

Proof. By Lemma 2, we know each M_i is either of the form $\mathsf{return}\, V$, or can be written $E[M']$ where M' is a communication or concurrency primitive. It cannot be the case that $M' = \mathsf{fork}\, N$ or $M' = \mathsf{newCh}$, since both can reduce. Let us now consider give and take, blocked on a variable α. As we are considering closed configurations, a blocked term must be blocked on a ν-bound name a_i, and as per the canonical form, we have that there exists some buffer $a_i(\overrightarrow{V_i})$. Consequently, $\mathsf{give}\, V\, a_i$ can always reduce via Give. A term $\mathsf{take}\, a_i$ can reduce by Take if $\overrightarrow{V_i} = W \cdot \overrightarrow{V_i'}$; the only remaining case is where $\overrightarrow{V_i} = \epsilon$, satisfying (3). ◀

4 λ_{act}: A concurrent λ-calculus for actors

In this section, we introduce λ_{act}, a core language describing actor-based concurrency. There are many variations of actor-based languages (by the taxonomy of De Koster et al; [11], λ_{act} is *process-based*), but each have named processes associated with a mailbox.

Typed channels are well-established, whereas typed actors are less so, partly due to the type pollution problem. Nonetheless, Akka Typed [4] aims to replace untyped Akka actors, so studying a typed actor calculus is of practical relevance.

Following Erlang, we provide an explicit receive operation to allow an actor to retrieve a message from its mailbox: unlike take in λ_{ch}, receive takes no arguments, so it is necessary to use a simple *type-and-effect system* [18]. We treat mailboxes as a FIFO queues to keep λ_{act} as minimal as possible, as opposed to considering behaviours or selective receive. This is orthogonal to the core model of communication, as we show in Section 7.2.

Syntax

$$
\begin{array}{lrl}
\text{Types} & A, B, C & ::= \mathbf{1} \mid A \to^{C} B \mid \mathsf{ActorRef}(A) \\
\text{Variables and names} & \alpha & ::= x \mid a \\
\text{Values} & V, W & ::= \alpha \mid \lambda x.M \mid () \\
\text{Computations} & L, M, N & ::= V\,W \\
 & & \mid \mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N \mid \mathsf{return}\ V \\
 & & \mid \mathsf{spawn}\ M \mid \mathsf{send}\ V\,W \mid \mathsf{receive} \mid \mathsf{self}
\end{array}
$$

Value typing rules $\boxed{\Gamma \vdash V : A}$

$$
\textsc{Var}\ \frac{\alpha : A \in \Gamma}{\Gamma \vdash \alpha : A} \qquad \textsc{Abs}\ \frac{\Gamma, x : A \mid C \vdash M : B}{\Gamma \vdash \lambda x.M : A \to^{C} B} \qquad \textsc{Unit}\ \frac{}{\Gamma \vdash () : \mathbf{1}}
$$

Computation typing rules $\boxed{\Gamma \mid B \vdash M : A}$

$$
\textsc{App}\ \frac{\Gamma \vdash V : A \to^{C} B \quad \Gamma \vdash W : A}{\Gamma \mid C \vdash V\,W : B} \qquad \textsc{EffLet}\ \frac{\Gamma \mid C \vdash M : A \quad \Gamma, x : A \mid C \vdash N : B}{\Gamma \mid C \vdash \mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N : B}
$$

$$
\textsc{EffReturn}\ \frac{\Gamma \vdash V : A}{\Gamma \mid C \vdash \mathsf{return}\ V : A} \qquad \textsc{Send}\ \frac{\Gamma \vdash V : A \quad \Gamma \vdash W : \mathsf{ActorRef}(A)}{\Gamma \mid C \vdash \mathsf{send}\ V\,W : \mathbf{1}}
$$

$$
\textsc{Recv}\ \frac{}{\Gamma \mid A \vdash \mathsf{receive} : A} \qquad \textsc{Spawn}\ \frac{\Gamma \mid A \vdash M : \mathbf{1}}{\Gamma \mid C \vdash \mathsf{spawn}\ M : \mathsf{ActorRef}(A)} \qquad \textsc{Self}\ \frac{}{\Gamma \mid A \vdash \mathsf{self} : \mathsf{ActorRef}(A)}
$$

Figure 8 Syntax and typing rules for λ_{act}.

4.1 Syntax and typing of terms

Figure 8 shows the syntax and typing rules for λ_{act}. As with λ_{ch}, α ranges over variables and names. $\mathsf{ActorRef}(A)$ is an *actor reference* or process ID, and allows messages to be sent to an actor. As for communication and concurrency primitives, $\mathsf{spawn}\ M$ spawns a new actor to evaluate a computation M; $\mathsf{send}\ V\ W$ sends a value V to an actor referred to by reference W; $\mathsf{receive}$ receives a value from the actor's mailbox; and self returns an actor's own process ID.

Function arrows $A \to^{C} B$ are annotated with a type C which denotes the type of the mailbox of the actor evaluating the term. As an example, consider a function which receives an integer and converts it to a string (assuming a function `intToString`):

$$\texttt{recvAndShow} \triangleq \lambda().\mathsf{let}\ x \Leftarrow \mathsf{receive}\ \mathsf{in}\ \texttt{intToString}(x)$$

Such a function would have type $\mathbf{1} \to^{Int} \mathit{String}$, and as an example would not be typeable for an actor that could only receive booleans. Again, we work in the setting of fine-grain call-by-value; the distinction between values and computations is helpful when reasoning about the metatheory. We have two typing judgements: the standard judgement on values $\Gamma \vdash V : A$, and a judgement $\Gamma \mid B \vdash M : A$ which states that a term M has type A under typing context Γ, and can receive values of type B. The typing of $\mathsf{receive}$ and self depends on the type of the actor's mailbox.

4.2 Operational semantics

Figure 9 shows the syntax of λ_{act} evaluation contexts, as well as the syntax and typing rules of λ_{act} configurations. Evaluation contexts for terms and configurations are similar to λ_{ch}. The primary difference from λ_{ch} is the actor configuration $\langle a, M, \overrightarrow{V} \rangle$, which can be read as

Syntax of evaluation contexts and configurations

Evaluation contexts	E	$::= [\,] \mid \mathsf{let}\ x \Leftarrow E\ \mathsf{in}\ M$
Configurations	$\mathcal{C}, \mathcal{D}, \mathcal{E}$	$::= \mathcal{C} \parallel \mathcal{D} \mid (\nu a)\mathcal{C} \mid \langle a, M, \overrightarrow{V} \rangle$
Configuration contexts	G	$::= [\,] \mid G \parallel \mathcal{C} \mid (\nu a)G$

Typing rules for configurations $\boxed{\Gamma; \Delta \vdash \mathcal{C}}$

$$\textsc{Par}\ \frac{\Gamma; \Delta_1 \vdash \mathcal{C}_1 \quad \Gamma; \Delta_2 \vdash \mathcal{C}_2}{\Gamma; \Delta_1, \Delta_2 \vdash \mathcal{C}_1 \parallel \mathcal{C}_2} \qquad \textsc{Pid}\ \frac{\Gamma, a : \mathsf{ActorRef}(A); \Delta, a : A \vdash \mathcal{C}}{\Gamma; \Delta \vdash (\nu a)\mathcal{C}} \qquad \textsc{Actor}\ \frac{\Gamma, a : \mathsf{ActorRef}(A) \mid A \vdash M : \mathbf{1} \quad (\Gamma, a : \mathsf{ActorRef}(A) \vdash V_i : A)_i}{\Gamma, a : \mathsf{ActorRef}(A); a : A \vdash \langle a, M, \overrightarrow{V} \rangle}$$

Figure 9 λ_{act} evaluation contexts and configurations.

"an actor with name a evaluating term M, with a mailbox consisting of values $\overrightarrow{V}$". Whereas a term M is itself a configuration in λ_{ch}, a term in λ_{act} must be evaluated as part of an actor configuration in order to support context-sensitive operations such as receiving from the mailbox. We again stratify the reduction rules into functional reduction on terms, and reduction on configurations. The typing rules for λ_{act} configurations ensure that all values contained in an actor mailbox are well-typed with respect to the mailbox type, and that a configuration $\mathcal{C}$ under a name restriction $(\nu a)\mathcal{C}$ contains an actor with name a. Figure 10 shows the reduction rules for λ_{act}. Again, reduction on terms preserves typing, and the functional fragment of λ_{act} enjoys progress.

▶ **Lemma 9** (Preservation (λ_{act} terms)). *If $\Gamma \vdash M : A$ and $M \longrightarrow_M M'$, then $\Gamma \vdash M' : A$.*

▶ **Lemma 10** (Progress (λ_{act} terms)). *Assume Γ is either empty or only contains entries of the form $a_i : \mathsf{ActorRef}(A_i)$. If $\Gamma \mid B \vdash M : A$, then either:*

1. *$M = \mathsf{return}\, V$ for some value V, or*
2. *M can be written as $E[M']$, where M' is a communication or concurrency primitive (i.e. $\mathsf{spawn}\, N$, $\mathsf{send}\, V\, W$, $\mathsf{receive}$, or self), or*
3. *There exists some M' such that $M \longrightarrow_M M'$.*

Reduction on configurations. While λ_{ch} makes use of separate constructs to create new processes and channels, λ_{act} uses a single construct $\mathsf{spawn}\, M$ to spawn a new actor with an empty mailbox to evaluate term M. Communication happens directly between actors instead of through an intermediate entity: as a result of evaluating $\mathsf{send}\, V\, a$, the value V will be appended directly to the end of the mailbox of actor a. SendSelf allows reflexive sending; an alternative would be to decouple mailboxes from the definition of actors, but this complicates both the configuration typing rules and the intuition. Self returns the name of the current process, and Receive retrieves the head value of a non-empty mailbox.

As before, typing is preserved modulo structural congruence and under reduction.

▶ **Lemma 11.** *If $\Gamma; \Delta \vdash \mathcal{C}$ and $\mathcal{C} \equiv \mathcal{D}$ for some $\mathcal{D}$, then $\Gamma; \Delta \vdash \mathcal{D}$.*

▶ **Theorem 12** (Preservation (λ_{act} configurations)). *If $\Gamma; \Delta \vdash \mathcal{C}_1$ and $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$, then $\Gamma; \Delta \vdash \mathcal{C}_2$.*

4.3 Progress and canonical forms

Again, we cannot guarantee deadlock-freedom for λ_{act}. Instead, we proceed by defining a canonical form, and characterising the form of progress that λ_{act} enjoys. The technical development follows that of λ_{ch}.

Reduction on terms

$$(\lambda x.M)\,V \longrightarrow_{\mathsf{M}} M\{V/x\} \qquad \mathsf{let}\ x \Leftarrow \mathsf{return}\ V\ \mathsf{in}\ M \longrightarrow_{\mathsf{M}} M\{V/x\} \qquad E[M] \longrightarrow_{\mathsf{M}} E[M'] \ (\text{if } M \longrightarrow_{\mathsf{M}} M')$$

Structural congruence

$$\mathcal{C} \parallel \mathcal{D} \equiv \mathcal{D} \parallel \mathcal{C} \qquad \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) \equiv (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E} \qquad \mathcal{C} \parallel (\nu a)\mathcal{D} \equiv (\nu a)(\mathcal{C} \parallel \mathcal{D}) \text{ if } a \notin \mathsf{fv}(\mathcal{C})$$

$$G[\mathcal{C}] \equiv G[\mathcal{D}] \text{ if } \mathcal{C} \equiv \mathcal{D}$$

Reduction on configurations

$$\begin{array}{rrcl}
\textsc{Spawn} & \langle a, E[\mathsf{spawn}\ M], \overrightarrow{V}\rangle & \longrightarrow & (\nu b)(\langle a, E[\mathsf{return}\ b], \overrightarrow{V}\rangle \parallel \langle b, M, \epsilon\rangle) \quad (b \text{ is fresh}) \\
\textsc{Send} & \langle a, E[\mathsf{send}\ V'\ b], \overrightarrow{V}\rangle \parallel \langle b, M, \overrightarrow{W}\rangle & \longrightarrow & \langle a, E[\mathsf{return}\ ()], \overrightarrow{V}\rangle \parallel \langle b, M, \overrightarrow{W} \cdot V'\rangle \\
\textsc{SendSelf} & \langle a, E[\mathsf{send}\ V'\ a], \overrightarrow{V}\rangle & \longrightarrow & \langle a, E[\mathsf{return}\ ()], \overrightarrow{V} \cdot V'\rangle \\
\textsc{Self} & \langle a, E[\mathsf{self}], \overrightarrow{V}\rangle & \longrightarrow & \langle a, E[\mathsf{return}\ a], \overrightarrow{V}\rangle \\
\textsc{Receive} & \langle a, E[\mathsf{receive}], W \cdot \overrightarrow{V}\rangle & \longrightarrow & \langle a, E[\mathsf{return}\ W], \overrightarrow{V}\rangle \\
\textsc{Lift} & G[\mathcal{C}_1] & \longrightarrow & G[\mathcal{C}_2] \quad (\text{if } \mathcal{C}_1 \longrightarrow \mathcal{C}_2) \\
\textsc{LiftM} & \langle a, M_1, \overrightarrow{V}\rangle & \longrightarrow & \langle a, M_2, \overrightarrow{V}\rangle \quad (\text{if } M_1 \longrightarrow_{\mathsf{M}} M_2)
\end{array}$$

Figure 10 Reduction on λ_{act} terms and configurations.

▶ **Definition 13** (Canonical form (λ_{act})). A λ_{act} configuration $\mathcal{C}$ is in *canonical form* if $\mathcal{C}$ can be written $(\nu a_1)\ldots(\nu a_n)(\langle a_1, M_1, \overrightarrow{V_1}\rangle \parallel \ldots \parallel \langle a_n, M_n, \overrightarrow{V_n}\rangle)$.

▶ **Lemma 14.** *If $\Gamma; \Delta \vdash \mathcal{C}$ and $\Delta = a_1 : A_1, \ldots a_k : A_k$, then there exists $\mathcal{C}' \equiv \mathcal{C}$ such that $\mathcal{C}' = (\nu a_{k+1})\ldots(\nu a_n)(\langle a_1, M_1, \overrightarrow{V_1}\rangle \parallel \ldots \parallel \langle a_n, M_n, \overrightarrow{V_n}\rangle)$.*

As before, it follows as a corollary of Lemma 14 that closed configurations can be written in canonical form. We can therefore classify the notion of progress enjoyed by λ_{act}.

▶ **Corollary 15.** *If $\cdot; \cdot \vdash \mathcal{C}$, then there exists some $\mathcal{C}' \equiv \mathcal{C}$ such that $\mathcal{C}'$ is in canonical form.*

▶ **Theorem 16** (Weak progress (λ_{act} configurations)). *Let $\cdot; \cdot \vdash \mathcal{C}$, $\mathcal{C} \not\longrightarrow$, and let $\mathcal{C}' = (\nu a_1)\ldots(\nu a_n)(\langle a_1, M_1, \overrightarrow{V_1}\rangle \parallel \ldots \parallel \langle a_n, M_n, \overrightarrow{V_n}\rangle)$ be a canonical form of $\mathcal{C}$. Each actor with name a_i is either of the form $\langle a_i, \mathsf{return}\ W, \overrightarrow{V_i}\rangle$ for some value W, or $\langle a_i, E[\mathsf{receive}], \epsilon\rangle$.*

5 From λ_{act} to λ_{ch}

With both calculi in place, we can define the translation from λ_{act} into λ_{ch}. The key idea is to emulate a mailbox using a channel, and to pass the channel as an argument to each function. The translation on terms is parameterised over the channel name, which is used to implement context-dependent operations (i.e., receive and self). Consider again **recvAndShow**.

$$\mathtt{recvAndShow} \triangleq \lambda().\mathsf{let}\ x \Leftarrow \mathsf{receive}\ \mathsf{in}\ \mathtt{intToString}(x)$$

A possible configuration would be an actor evaluating **recvAndShow** (), with some name a and mailbox with values $\overrightarrow{V}$, under a name restriction for a.

$$(\nu a)(\langle a, \mathtt{recvAndShow}\,(), \overrightarrow{V}\rangle)$$

Translation on types

$$[\![\,\mathsf{ActorRef}(A)\,]\!] = \mathsf{ChanRef}([\![\,A\,]\!]) \qquad [\![\,A \to^{C} B\,]\!] = [\![\,A\,]\!] \to \mathsf{ChanRef}([\![\,C\,]\!]) \to [\![\,B\,]\!] \qquad [\![\,\mathbf{1}\,]\!] = \mathbf{1}$$

Translation on values

$$[\![\,x\,]\!] = x \qquad [\![\,a\,]\!] = a \qquad [\![\,\lambda x.M\,]\!] = \lambda x.\lambda ch.([\![\,M\,]\!]\ ch) \qquad [\![\,()\,]\!] = ()$$

Translation on computation terms

$$[\![\,\mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N\,]\!]\ ch = \mathsf{let}\ x \Leftarrow ([\![\,M\,]\!]\ ch)\ \mathsf{in}\ [\![\,N\,]\!]\ ch$$

$$\begin{array}{rcl@{\qquad}rcl}
[\![\,V\ W\,]\!]\ ch & = & \mathsf{let}\ f \Leftarrow ([\![\,V\,]\!]\,[\![\,W\,]\!])\ \mathsf{in}\ f\ ch & [\![\,\mathsf{spawn}\ M\,]\!]\ ch & = & \mathsf{let}\ chMb \Leftarrow \mathsf{newCh}\ \mathsf{in} \\
[\![\,\mathsf{return}\ V\,]\!]\ ch & = & \mathsf{return}\ [\![\,V\,]\!] & & & \mathsf{fork}\ ([\![\,M\,]\!]\ chMb); \\
[\![\,\mathsf{self}\,]\!]\ ch & = & \mathsf{return}\ ch & & & \mathsf{return}\ chMb \\
[\![\,\mathsf{receive}\,]\!]\ ch & = & \mathsf{take}\ ch & [\![\,\mathsf{send}\ V\ W\,]\!]\ ch & = & \mathsf{give}\ ([\![\,V\,]\!])\ ([\![\,W\,]\!])
\end{array}$$

Translation on configurations

$$[\![\,\mathcal{C}_1 \parallel \mathcal{C}_2\,]\!] = [\![\,\mathcal{C}_1\,]\!] \parallel [\![\,\mathcal{C}_2\,]\!] \qquad [\![\,(\nu a)\mathcal{C}\,]\!] = (\nu a)\,[\![\,\mathcal{C}\,]\!] \qquad [\![\,\langle a, M, \overrightarrow{V}\rangle\,]\!] = a([\![\,\overrightarrow{V}\,]\!]) \parallel ([\![\,M\,]\!]\ a)$$

Figure 11 Translation from λ_{act} into λ_{ch}.

The translation on terms takes a channel name ch as a parameter. As a result of the translation, we have that:

$$[\![\,\mathtt{recvAndShow}\,()\,]\!]\ ch = \mathsf{let}\ x \Leftarrow \mathsf{take}\ ch\ \mathsf{in}\ \mathtt{intToString}(x)$$

with the corresponding configuration $(\nu a)(a([\![\,\overrightarrow{V}\,]\!]) \parallel [\![\,\mathtt{recvAndShow}\,()\,]\!]\ a)$. The values from the mailbox are translated pointwise and form the contents of a buffer with name a. The translation of `recvAndShow` is provided with the name a which is used to emulate receive.

5.1 Translation (λ_{act} to λ_{ch})

Figure 11 shows the formal translation from λ_{act} into λ_{ch}. Of particular note is the translation on terms: $[\![\,-\,]\!]\ ch$ translates a λ_{act} term into a λ_{ch} term using a channel with name ch to emulate a mailbox. An actor reference is represented as a channel reference in λ_{ch}; we emulate sending a message to another actor by writing to the channel emulating the recipient's mailbox. Key to translating λ_{act} into λ_{ch} is the translation of function arrows $A \to^{C} B$; the effect annotation C is replaced by a second parameter $\mathsf{ChanRef}(C)$, which is used to emulate the mailbox of the actor. Values translate to themselves, with the exception of λ abstractions, whose translation takes an additional parameter denoting the channel used to emulate operations on a mailbox. Given parameter ch, the translation function for terms emulates receive by taking a value from ch, and emulates self by returning ch.

Though the translation is straightforward, it is a *global* translation [12], as all functions must be modified in order to take the mailbox channel as an additional parameter.

5.2 Properties of the translation

The translation on terms and values preserves typing. We extend the translation function pointwise to typing environments: $[\![\,\alpha_1 : A_1, \ldots, \alpha_n : A_n\,]\!] = \alpha_1 : [\![\,A_1\,]\!], \ldots, \alpha_n : [\![\,A_n\,]\!]$.

▶ **Lemma 17** ($[\![\,-\,]\!]$ preserves typing (terms and values)).

1. *If* $\Gamma \vdash V : A$ *in* λ_{act}, *then* $[\![\,\Gamma\,]\!] \vdash [\![\,V\,]\!] : [\![\,A\,]\!]$ *in* λ_{ch}.

2. If $\Gamma \mid B \vdash M : A$ *in* λ_{act}*, then* $[\![\,\Gamma\,]\!], \alpha : \mathsf{ChanRef}([\![\,B\,]\!]) \vdash [\![\,M\,]\!]\ \alpha : [\![\,A\,]\!]$ *in* λ_{ch}*.*

The proof is by simultaneous induction on the derivations of $\Gamma \vdash V{:}A$ and $\Gamma \mid B \vdash M{:}A$. To state a semantics preservation result, we also define a translation on configurations; the translations on parallel composition and name restrictions are homomorphic. An actor configuration $\langle a, M, \overrightarrow{V}\rangle$ is translated as a buffer $a([\![\,\overrightarrow{V}\,]\!])$, (writing $[\![\,\overrightarrow{V}\,]\!] = [\![\,V_0\,]\!]\cdot, \ldots, \cdot[\![\,V_n\,]\!]$ for each $V_i \in \overrightarrow{V}$), composed in parallel with the translation of M, using a as the mailbox channel. We can now see that the translation preserves typeability of configurations.

▶ **Theorem 18** ($[\![-]\!]$ preserves typeability (configurations))**.**
If $\Gamma; \Delta \vdash \mathcal{C}$ *in* λ_{act}*, then* $[\![\,\Gamma\,]\!]; [\![\,\Delta\,]\!] \vdash [\![\,\mathcal{C}\,]\!]$ *in* λ_{ch}*.*

We describe semantics preservation in terms of a simulation theorem: should a configuration $\mathcal{C}_1$ reduce to a configuration $\mathcal{C}_2$ in λ_{act}, then there exists some configuration $\mathcal{D}$ in λ_{ch} such that $[\![\,\mathcal{C}_1\,]\!]$ reduces in zero or more steps to $\mathcal{D}$, with $\mathcal{D} \equiv [\![\,\mathcal{C}_2\,]\!]$. To establish the result, we begin by showing that λ_{act} term reduction can be simulated in λ_{ch}.

▶ **Lemma 19** (Simulation of λ_{act} term reduction in λ_{ch})**.**
If $\Gamma \vdash M_1 : A$ *and* $M_1 \longrightarrow_M M_2$ *in* λ_{act}*, then given some* α*,* $[\![\,M_1\,]\!]\ \alpha \longrightarrow_M^* [\![\,M_2\,]\!]\ \alpha$ *in* λ_{ch}*.*

Finally, we can see that the translation preserves structural congruences, and that λ_{ch} configurations can simulate reductions in λ_{act}.

▶ **Lemma 20.** *If* $\Gamma; \Delta \vdash \mathcal{C}$ *and* $\mathcal{C} \equiv \mathcal{D}$*, then* $[\![\,\mathcal{C}\,]\!] \equiv [\![\,\mathcal{D}\,]\!]$*.*

▶ **Theorem 21** (Simulation of λ_{act} configurations in λ_{ch})**.**
If $\Gamma; \Delta \vdash \mathcal{C}_1$ *and* $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$*, then there exists some* $\mathcal{D}$ *such that* $[\![\,\mathcal{C}_1\,]\!] \longrightarrow^* \mathcal{D}$*, with* $\mathcal{D} \equiv [\![\,\mathcal{C}_2\,]\!]$*.*

6 From λ_{ch} to λ_{act}

The translation from λ_{act} into λ_{ch} emulates an actor mailbox using a channel to implement operations which normally rely on the context of the actor. Though global, the translation is straightforward due to the limited form of communication supported by mailboxes. Translating from λ_{ch} into λ_{act} is more challenging, as would be expected from Figure 2. Each channel in a system may have a different type; each process may have access to multiple channels; and (crucially) channels may be freely passed between processes.

6.1 Extensions to the core language

We require several more language constructs: sums, products, recursive functions, and iso-recursive types. Recursive functions are used to implement an event loop, and recursive types to maintain a term-level buffer. Products are used to record both a list of values in the buffer and a list of pending requests. Sum types allow the disambiguation of the two types of messages sent to an actor: one to queue a value (emulating give) and one to dequeue a value (emulating take). Sums are also used to encode monomorphic variant types; we write $\langle \ell_1 : A_1, \ldots, \ell_n : A_n \rangle$ for variant types and $\langle \ell_i = V \rangle$ for variant values.

Figure 12 shows the extensions to the core term language and their reduction rules; we omit the symmetric rules for inr. With products, sums, and recursive types, we can encode lists. The typing rules are shown for λ_{ch} but can be easily adapted for λ_{act}, and it is straightforward to verify that the extended languages still enjoy progress and preservation.

Syntax

Types $A, B, C ::= \ldots \mid A \times B \mid A + B \mid \mathsf{List}(A) \mid \mu X.A \mid X$

Values $V, W ::= \ldots \mid \mathsf{rec}\, f(x)\,.\,M \mid (V, W) \mid \mathsf{inl}\, V \mid \mathsf{inr}\, W \mid \mathsf{roll}\, V$

Terms $L, M, N ::= \ldots \mid \mathsf{let}\,(x, y) = V\,\mathsf{in}\,M \mid \mathsf{case}\, V\, \{\mathsf{inl}\, x \mapsto M; \mathsf{inr}\, y \mapsto N\} \mid \mathsf{unroll}\, V$

Additional value typing rules $\boxed{\Gamma \vdash V : A}$

$$\textsc{Rec}\ \frac{\Gamma, x : A, f : A \to B \vdash M : B}{\Gamma \vdash \mathsf{rec}\, f(x)\,.\,M : A \to B} \qquad \textsc{Pair}\ \frac{\Gamma \vdash V : A \quad \Gamma \vdash W : B}{\Gamma \vdash (V, W) : A \times B} \qquad \textsc{Inl}\ \frac{\Gamma \vdash V : A}{\Gamma \vdash \mathsf{inl}\, V : A + B} \qquad \textsc{Roll}\ \frac{\Gamma \vdash V : A\{\mu X.A/X\}}{\Gamma \vdash \mathsf{roll}\, V : \mu X.A}$$

Additional term typing rules $\boxed{\Gamma \vdash M : A}$

$$\textsc{Let}\ \frac{\Gamma \vdash V : A \times A' \quad \Gamma, x : A, y : A' \vdash M : B}{\Gamma \vdash \mathsf{let}\,(x, y) = V\,\mathsf{in}\,M : B} \qquad \textsc{Case}\ \frac{\Gamma \vdash V : A + A' \quad \Gamma, x : A \vdash M : B \quad \Gamma, y : A' \vdash N : B}{\Gamma \vdash \mathsf{case}\, V\, \{\mathsf{inl}\, x \mapsto M; \mathsf{inr}\, y \mapsto N\} : B}$$

$$\textsc{Unroll}\ \frac{\Gamma \vdash V : \mu X.A}{\Gamma \vdash \mathsf{unroll}\, V : A\{\mu X.A/X\}}$$

Additional term reduction rules $\boxed{M \longrightarrow_{\mathsf{M}} M'}$

$$\begin{aligned}
(\mathsf{rec}\, f(x)\,.\,M)\, V &\longrightarrow_{\mathsf{M}} M\{(\mathsf{rec}\, f(x)\,.\,M)/f, V/x\} \\
\mathsf{let}\,(x, y) = (V, W)\,\mathsf{in}\,M &\longrightarrow_{\mathsf{M}} M\{V/x, W/y\} \\
\mathsf{case}\,(\mathsf{inl}\, V)\, \{\mathsf{inl}\, x \mapsto M; \mathsf{inr}\, y \mapsto N\} &\longrightarrow_{\mathsf{M}} M\{V/x\} \\
\mathsf{unroll}\,(\mathsf{roll}\, V) &\longrightarrow_{\mathsf{M}} \mathsf{return}\, V
\end{aligned}$$

Encoding of lists

$$\mathsf{List}(A) \triangleq \mu X.1 + (A \times X) \qquad [\,] \triangleq \mathsf{roll}\,(\mathsf{inl}\,()) \qquad V :: W \triangleq \mathsf{roll}\,(\mathsf{inr}\,(V, W))$$

$$\mathsf{case}\, V\, \{[\,] \mapsto M; x :: y \mapsto N\} \triangleq \mathsf{let}\, z \Leftarrow \mathsf{unroll}\, V\, \mathsf{in}\, \mathsf{case}\, z\, \{\mathsf{inl}\,() \mapsto M; \mathsf{inr}\,(x, y) \mapsto N\}$$

Figure 12 Extensions to core languages to allow translation from λ_{ch} into λ_{act}.

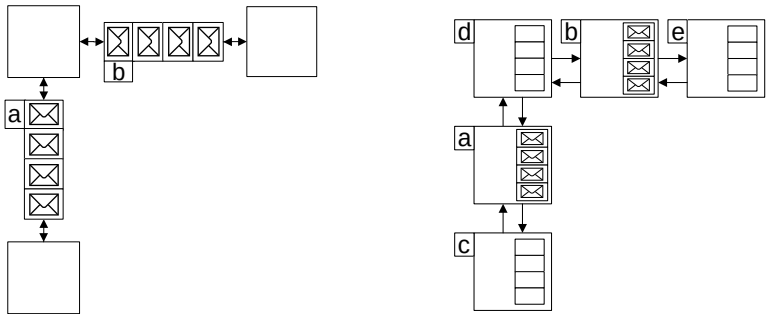
6.2 Translation strategy (λ_{ch} into λ_{act})

To translate typed actors into typed channels (shown in Figure 13), we emulate each channel using an actor process, which is crucial in retaining the mobility of channel endpoints. Channel types describe the typing of a *communication medium* between communicating processes, where processes are unaware of the identity of other communicating parties, and the types of messages that another party may receive. Unfortunately, the same does not hold for mailboxes. Consequently, we require that before translating into actors, *every channel has the same type*. Although this may seem restrictive, it is both possible and safe to transform a λ_{ch} program with multiple channel types into a λ_{ch} program with a single channel type.

As an example, suppose we have a program which contains channels carrying values of types Int, String, and $\mathsf{ChanRef}(\mathsf{String})$. It is possible to construct a recursive variant type $\mu X.\langle \ell_1 : \mathsf{Int}, \ell_2 : \mathsf{String}, \ell_3 : \mathsf{ChanRef}(X)\rangle$ which can be assigned to all channels in the system. Then, supposing we wanted to send a 5 along a channel which previously had type $\mathsf{ChanRef}(\mathsf{Int})$, we would instead send a value $\mathsf{roll}\,\langle \ell_1 = 5\rangle$ (where $\mathsf{roll}\, V$ is the introduction rule for an iso-recursive type). Appendix A [15] provides more details.

6.3 Translation

We write λ_{ch} judgements of the form $\{B\}\ \Gamma \vdash M : A$ for a term where all channels have type B, and similarly for value and configuration typing judgements. Under such a judgement, we can write Chan instead of $\mathsf{ChanRef}(B)$.



(a) Before Translation. **(b)** After Translation.

Figure 13 Translation strategy: λ_{ch} into λ_{act}.

Meta level definitions. The majority of the translation lies within the translation of newCh, which makes use of the meta-level definitions **`body`** and **`drain`**. The **`body`** function emulates a channel. Firstly, the actor receives a message *recvVal*, which is either of the form $\mathsf{inl}\, V$ to store a message V, or $\mathsf{inr}\, W$ to request that a value is dequeued and sent to the actor with ID W. We assume a standard implementation of list concatenation ($+\!\!+$). If the message is $\mathsf{inl}\, V$, then V is appended to the tail of the list of values stored in the channel, and the new state is passed as an argument to **`drain`**. If the message is $\mathsf{inr}\, W$, then the process ID W is appended to the end of the list of processes waiting for a value. The **`drain`** function satisfies all requests that can be satisfied, returning an updated channel state. Note that **`drain`** does not need to be recursive, since one of the lists will either be empty or a singleton.

Translation on types. Figure 14 shows the translation from λ_{ch} into λ_{act}. The translation function on types $(\!|-|\!)$ is defined with respect to the type of all channels C and is used to annotate function arrows and to assign a parameter to ActorRef types. The (omitted) translations on sums, products, and lists are homomorphic. The translation of Chan is $\mathsf{ActorRef}((\!|C|\!)+\mathsf{ActorRef}((\!|C|\!)))$, meaning an actor which can receive a request to either store a value of type $(\!|C|\!)$, or to dequeue a value and send it to a process ID of type $\mathsf{ActorRef}((\!|C|\!))$.

Translation on communication and concurrency primitives. We omit the translation on values and functional terms, which are homomorphisms. Processes in λ_{ch} are anonymous, whereas all actors in λ_{act} are addressable; to emulate fork, we therefore discard the reference returned by spawn. The translation of give wraps the translated value to be sent in the left injection of a sum type, and sends to the translated channel name $(\!|W|\!)$. To emulate take, the process ID (retrieved using self) is wrapped in the right injection and sent to the actor emulating the channel, and the actor waits for the response message. Finally, the translation of newCh spawns a new actor to execute **`body`**.

Translation on configurations. The translation function $(\!|-|\!)$ is homomorphic on parallel composition and name restriction. Unlike λ_{ch}, a term cannot exist outwith an enclosing actor context in λ_{act}, so the translation of a process evaluating term M is an actor evaluating $(\!|M|\!)$ with some fresh name a and an empty mailbox, enclosed in a name restriction. A buffer is translated to an actor with an empty mailbox, evaluating **`body`** with a state containing the (term-level) list of values previously stored in the buffer.

Although the translation from λ_{ch} into λ_{act}, is much more verbose than the translation from λ_{act} to λ_{ch}, it is (once all channels have the same type) a *local transformation* [12].

Translation on types (wrt. a channel type C)

$$(\!|\,\mathsf{Chan}\,|\!) = \mathsf{ActorRef}((\!|\,C\,|\!) + \mathsf{ActorRef}((\!|\,C\,|\!))) \qquad (\!|\,A \to B\,|\!) = (\!|\,A\,|\!) \to^{(\!|\,C\,|\!)} (\!|\,B\,|\!)$$

Translation on communication and concurrency primitives

$$\begin{array}{rcl}
(\!|\,\mathsf{fork}\,M\,|\!) & = & \mathsf{let}\ x \Leftarrow \mathsf{spawn}\,(\!|\,M\,|\!)\ \mathsf{in}\ \mathsf{return}\ () \\
(\!|\,\mathsf{give}\,V\,W\,|\!) & = & \mathsf{send}\ (\mathsf{inl}\,(\!|\,V\,|\!))\,(\!|\,W\,|\!) \\
(\!|\,\mathsf{newCh}\,|\!) & = & \mathsf{spawn}\ (\mathtt{body}\ [\,],[\,])
\end{array}$$

$$\begin{array}{rcl}
(\!|\,\mathsf{take}\,V\,|\!) & = & \mathsf{let}\ \mathit{selfPid} \Leftarrow \mathsf{self}\ \mathsf{in} \\
 & & \mathsf{send}\ (\mathsf{inr}\ \mathit{selfPid})\,(\!|\,V\,|\!); \\
 & & \mathsf{receive}
\end{array}$$

Translation on configurations

$$(\!|\,\mathcal{C}_1 \parallel \mathcal{C}_2\,|\!) = (\!|\,\mathcal{C}_1\,|\!) \parallel (\!|\,\mathcal{C}_2\,|\!) \qquad (\!|\,(\nu a)\mathcal{C}\,|\!) = (\nu a)(\!|\,\mathcal{C}\,|\!) \qquad (\!|\,M\,|\!) = (\nu a)(\langle a, (\!|\,M\,|\!), \epsilon\rangle)\ \ a \text{ is a fresh name}$$

$$(\!|\,a(\overrightarrow{V})\,|\!) = \langle a, \mathtt{body}\ (\!|\,\overrightarrow{V}\,|\!), [\,]), \epsilon\rangle \quad \text{where } (\!|\,\overrightarrow{V}\,|\!) = (\!|\,V_0\,|\!) :: \ldots :: (\!|\,V_n\,|\!) :: [\,]$$

Meta level definitions

```
body ≜ rec g(state) .
  let recvVal ⇐ receive in
  let (vals, pids) = state in
  case recvVal {
      inl v ↦ let vals′ ⇐ vals ++ [v] in
              let state′ ⇐ drain (vals′, pids) in
              g (state′)
      inr pid ↦ let pids′ ⇐ pids ++ [pid] in
                let state′ ⇐ drain (vals, pids′) in
                g (state′)   }
```

```
drain ≜ λx.
  let (vals, pids) = x in
  case vals {
    [ ] ↦ return (vals, pids)
    v :: vs ↦
      case pids {
          [ ] ↦ return (vals, pids)
          pid :: pids ↦ send v pid;
                        return (vs, pids)
      }   }
```

Figure 14 Translation from λ_{ch} into λ_{act}.

6.4 Properties of the translation

Since all channels in the source language of the translation have the same type, we can assume that each entry in the codomain of Δ is the same type B.

▶ **Definition 22** (Translation of typing environments wrt. a channel type B).

1. If $\Gamma = \alpha_1{:}A_1, \ldots, \alpha_n : A_n$, define $(\!|\,\Gamma\,|\!) = \alpha_1 : (\!|\,A_1\,|\!), \ldots, \alpha_n : (\!|\,A_n\,|\!)$.
2. Given a $\Delta = a_1 : B, \ldots, a_n : B$, define $(\!|\,\Delta\,|\!) =$
   $a_1 : ((\!|\,B\,|\!) + \mathsf{ActorRef}((\!|\,B\,|\!))), \ldots, a_n : ((\!|\,B\,|\!) + \mathsf{ActorRef}((\!|\,B\,|\!)))$.

The translation on terms preserves typing.

▶ **Lemma 23** ($(\!|\,-\,|\!)$ preserves typing (terms and values)).

1. *If* $\{B\}\ \Gamma \vdash V{:}A$, *then* $(\!|\,\Gamma\,|\!) \vdash (\!|\,V\,|\!){:}(\!|\,A\,|\!)$.
2. *If* $\{B\}\ \Gamma \vdash M{:}A$, *then* $(\!|\,\Gamma\,|\!) \mid (\!|\,B\,|\!) \vdash (\!|\,M\,|\!){:}(\!|\,A\,|\!)$.

The translation on configurations also preserves typeability. We write $\Gamma \asymp \Delta$ if for each $a : A \in \Delta$, we have that $a : \mathsf{ChanRef}(A) \in \Gamma$; for closed configurations this is ensured by Chan. This is necessary since the typing rules for λ_{act} require that the local actor name is present in the term environment to ensure preservation in the presence of self, but there is no such restriction in λ_{ch}.

▶ **Theorem 24** ($(\!|\,-\,|\!)$ preserves typeability (configurations)).
If $\{A\}\ \Gamma; \Delta \vdash \mathcal{C}$ *with* $\Gamma \asymp \Delta$, *then* $(\!|\,\Gamma\,|\!); (\!|\,\Delta\,|\!) \vdash (\!|\,\mathcal{C}\,|\!)$.

It is clear that reduction on translated λ_{ch} terms can simulate reduction in λ_{act}.

▶ **Lemma 25.** *If* $\{B\}\ \Gamma \vdash M_1 : A$ *and* $M_1 \longrightarrow_M M_2$*, then* $(\!|\, M_1 \,|\!) \longrightarrow_M (\!|\, M_2 \,|\!)$*.*

Finally, we show that λ_{act} can simulate λ_{ch}.

▶ **Lemma 26.** *If* $\Gamma; \Delta \vdash \mathcal{C}$ *and* $\mathcal{C} \equiv \mathcal{D}$*, then* $(\!|\, \mathcal{C} \,|\!) \equiv (\!|\, \mathcal{D} \,|\!)$*.*

▶ **Theorem 27** (Simulation (λ_{act} configurations in λ_{ch}))**.**
If $\{A\}\ \Gamma; \Delta \vdash \mathcal{C}_1$*, and* $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$*, then there exists some* $\mathcal{D}$ *such that* $(\!|\, \mathcal{C}_1 \,|\!) \longrightarrow^* \mathcal{D}$ *with* $\mathcal{D} \equiv (\!|\, \mathcal{C}_2 \,|\!)$*.*

Remark. The translation from λ_{ch} into λ_{act} is more involved than the translation from λ_{act} into λ_{ch} due to the asymmetry shown in Figure 2. Mailbox types are less precise; generally taking the form of a large variant type.

Typical implementations of this translation use synchronisation mechanisms such as futures or shared memory (see Section 7.1); the implementation shown in the Hopac documentation uses ML references [1]. Given the ubiquity of these abstractions, we were surprised to discover that the additional expressive power of synchronisation is not *necessary*. Our original attempt at a synchronisation-free translation was type-directed. We were surprised to discover that the translation can be described so succinctly after factoring out the coalescing step, which precisely captures the type pollution problem.

7 Extensions

In this section, we discuss common extensions to channel- and actor-based languages. Firstly, we discuss synchronisation, which is ubiquitous in practical implementations of actor-inspired languages. Adding synchronisation simplifies the translation from channels to actors, and relaxes the restriction that all channels must have the same type. Secondly, we consider an extension with Erlang-style selective receive, and show how to encode it in λ_{act}. Thirdly, we discuss how to nondeterministically choose a message from a collection of possible sources, and finally, we discuss what the translations tell us about the nature of behavioural typing disciplines for actors. Establishing exactly how the latter two extensions fit into our framework is the subject of ongoing and future work.

7.1 Synchronisation

Although communicating with an actor via asynchronous message passing suffices for many purposes, implementing "call-response" style interactions can become cumbersome. Practical implementations such as Erlang and Akka implement some way of synchronising on a result: Erlang achieves this by generating a unique reference to send along with a request, *selectively receiving* from the mailbox to await a response tagged with the same unique reference. Another method of synchronisation embraced by the Active Object community [33, 10, 32] and Akka is to generate a *future variable* which is populated with the result of the call.

Figure 15 details an extension of λ_{act} with a synchronisation primitive, wait, which encodes a deliberately restrictive form of synchronisation capable of emulating futures. The key idea behind wait is it allows some actor a to block until an actor b evaluates to a value; this value is then returned directly to a, bypassing the mailbox. A variation of the wait primitive is implemented as part of the Links [9] concurrency model. This is but one of multiple ways of allowing synchronisation; first-class futures, shared reference cells, or selective receive can achieve a similar result. We discuss wait as it avoids the need for new configurations.

Additional types, terms, configuration reduction rule, and equivalence

$$\text{Types} ::= \mathsf{ActorRef}(A, B) \mid \ldots \qquad \text{Terms} ::= \mathsf{wait}\, V \mid \ldots$$

$$\langle a, E[\mathsf{wait}\, b], \overrightarrow{V}\rangle \parallel \langle b, \mathsf{return}\, V', \overrightarrow{W}\rangle \longrightarrow \langle a, E[\mathsf{return}\, V'], \overrightarrow{V}\rangle \parallel \langle b, \mathsf{return}\, V', \overrightarrow{W}\rangle$$
$$(\nu a)(\langle a, \mathsf{return}\, V, \overrightarrow{V}\rangle) \parallel \mathcal{C} \equiv \mathcal{C}$$

Modified typing rules for terms $\boxed{\Gamma \mid A, B \vdash M : A}$

$$\textsc{Sync-Spawn}\ \frac{\Gamma \mid A, B \vdash M : B}{\Gamma \mid C, C' \vdash \mathsf{spawn}\, M : \mathsf{ActorRef}(A, B)} \qquad \textsc{Sync-Wait}\ \frac{\Gamma \vdash V : \mathsf{ActorRef}(A, B)}{\Gamma \mid C, C' \vdash \mathsf{wait}\, V : B} \qquad \textsc{Sync-Self}\ \frac{}{\Gamma \mid A, B \vdash \mathsf{self} : \mathsf{ActorRef}(A, B)}$$

Modified typing rules for configurations $\boxed{\Gamma; \Delta \vdash \mathcal{C}}$

$$\textsc{Sync-Actor}\ \frac{\Gamma, a{:}\mathsf{ActorRef}(A, B) \vdash M{:}B \quad (\Gamma, a{:}\mathsf{ActorRef}(A, B) \vdash V_i{:}A)_i}{\Gamma, a : \mathsf{ActorRef}(A, B); a{:}(A, B) \vdash \langle a, M, \overrightarrow{V}\rangle} \qquad \textsc{Sync-Nu}\ \frac{\Gamma, a : \mathsf{ActorRef}(A, B); \Delta, a : (A, B) \vdash \mathcal{C}}{\Gamma; \Delta \vdash (\nu a)\mathcal{C}}$$

Modified translation

$$(\!|\, \mathsf{ChanRef}(A) \,|\!) = \mathsf{ActorRef}((\!|\, A \,|\!) + \mathsf{ActorRef}((\!|\, A \,|\!), (\!|\, A \,|\!)), \mathbf{1})$$
$$(\!|\, A \to B \,|\!) = (\!|\, A \,|\!) \to^{C,\mathbf{1}} (\!|\, B \,|\!)$$

$$\begin{aligned}(\!|\, \mathsf{take}\, V \,|\!) = \mathsf{let}\ &\mathit{requestorPid} \Leftarrow \mathsf{spawn}\ (\\ &\mathsf{let}\ \mathit{selfPid} \Leftarrow \mathsf{self}\ \mathsf{in}\\ &\mathsf{send}\ (\mathsf{inr}\ \mathit{selfPid})\ (\!|\, V \,|\!);\\ &\mathsf{receive})\ \mathsf{in}\\ &\mathsf{wait}\ \mathit{requestorPid}\end{aligned}$$

Figure 15 Extensions to add synchronisation to λ_{act}.

We replace the unary type constructor for process IDs with a binary type constructor $\mathsf{ActorRef}(A, B)$, where A is the type of messages that the process can receive from its mailbox, and B is the type of value to which the process will eventually evaluate. We assume that the remainder of the primitives are modified to take the additional effect type into account. We can now adapt the previous translation from λ_{ch} to λ_{act}, making use of wait to avoid the need for the coalescing transformation. Channel references are translated into actor references which can either receive a value of type A, or the PID of a process which can receive a value of type A and will eventually evaluate to a value of type A. Note that the unbound annotation $C, 1$ on function arrows reflects that the mailboxes can be of *any* type, since the mailboxes are unused in the actors emulating threads.

The key idea behind the modified translation is to spawn a fresh actor which makes the request to the channel and blocks waiting for the response. Once the spawned actor has received the result, the result can be retrieved synchronously using wait *without* reading from the mailbox. The previous soundness theorems adapt to the new setting.

▶ **Theorem 28.** *If* $\Gamma; \Delta \vdash \mathcal{C}$ *with* $\Gamma \asymp \Delta$, *then* $(\!|\, \Gamma \,|\!); (\!|\, \Delta \,|\!) \vdash (\!|\, \mathcal{C} \,|\!)$.

▶ **Theorem 29.** *If* $\Gamma; \Delta \vdash \mathcal{C}_1$ *and* $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$, *then there exists some* $\mathcal{D}$ *such that* $(\!|\, \mathcal{C} \,|\!) \longrightarrow^* \mathcal{D}$ *with* $\mathcal{D} \equiv (\!|\, \mathcal{C}_2 \,|\!)$.

The translation in the other direction requires named threads and a join construct in λ_{ch}.

Additional syntax

$$\begin{array}{rrcl} \text{Receive Patterns} & c & ::= & (\langle \ell = x \rangle \text{ when } M) \mapsto N \\ \text{Computations} & M & ::= & \text{receive } \{\overrightarrow{c}\} \mid \ldots \end{array}$$

Additional term typing rule

$$\textsc{Sel-Recv}\ \frac{\overrightarrow{c} = \{\langle \ell_i = x_i \rangle \text{ when } M_i \mapsto N_i\}_i \quad i \in J \qquad \Gamma, x_i : A_i \vdash_{\mathsf{P}} M_i : \mathsf{Bool} \quad \Gamma, x_i : A_i \mid \langle \ell_j : A_j \rangle_{j \in J} \vdash N_i : C}{\Gamma \mid \langle \ell_j : A_j \rangle_{j \in J} \vdash \text{receive } \{\overrightarrow{c}\} : C}$$

Additional configuration reduction rule

$$\frac{\exists k, l. \forall i. i < k \Rightarrow \neg(\mathsf{matchesAny}(\overrightarrow{c}, V_i)) \wedge \mathsf{matches}(c_l, V_k) \wedge \forall j. j < l \Rightarrow \neg(\mathsf{matches}(c_j, V_k))}{\langle a, E[\text{receive } \{\overrightarrow{c}\}], \overrightarrow{W} \cdot V_k \cdot \overrightarrow{W'} \rangle \longrightarrow \langle a, E[N_l\{V'_k/x_l\}], \overrightarrow{W} \cdot \overrightarrow{W'} \rangle}$$

where

$$\overrightarrow{c} = \{\langle \ell_i = x_i \rangle \text{ when } M_i \mapsto N_i\}_i \qquad \overrightarrow{W} = V_1 \cdot \ldots \cdot V_{k-1} \qquad \overrightarrow{W'} = V_{k+1} \cdot \ldots \cdot V_n \qquad V_k = \langle \ell_k = V'_k \rangle$$

$$\mathsf{matches}((\langle \ell = x \rangle \text{ when } M) \mapsto N, \langle \ell' = V \rangle) \triangleq (\ell = \ell') \wedge (M\{V/x\} \longrightarrow^*_{\mathsf{M}} \text{return } \mathsf{true})$$

$$\mathsf{matchesAny}(\overrightarrow{c}, V) \triangleq \exists c \in \overrightarrow{c}.\mathsf{matches}(c, V)$$

Figure 16 Additional syntax, typing rules, and reduction rules for λ_{act} with selective receive.

7.2 Selective receive

The receive construct in λ_{act} can only read the first message in the queue, which is cumbersome as it often only makes sense for an actor to handle a subset of messages at a given time.

In practice, Erlang provides a *selective receive* construct, matching messages in the mailbox against multiple pattern clauses. Assume we have a mailbox containing values $V_1, \ldots V_n$ and evaluate receive $\{c_1, \ldots, c_m\}$. The construct first tries to match value V_1 against clause c_1—if it matches, then the body of c_1 is evaluated, whereas if it fails, V_1 is tested against c_2 and so on. Should V_1 not match any pattern, then the process is repeated until V_n has been tested against c_m. At this point, the process blocks until a matching message arrives.

More concretely, consider an actor with mailbox type $C = \langle \mathsf{PriorityMessage} : \mathsf{Message}, \mathsf{StandardMessage} : \mathsf{Message}, \mathsf{Timeout} : \mathbf{1} \rangle$ which can receive both high- and low-priority messages. Let *getPriority* be a function which extracts a priority from a message.

Now consider the following actor:

```
receive {
    ⟨PriorityMessage = msg⟩ when (getPriority msg) > 5 ↦ handleMessage msg
    ⟨Timeout = msg⟩         when true                  ↦ ()
};
receive {
    ⟨PriorityMessage = msg⟩  when true ↦ handleMessage msg
    ⟨StandardMessage = msg⟩  when true ↦ handleMessage msg
    ⟨Timeout = msg⟩          when true ↦ ()
}
```

This actor begins by handling a message only if it has a priority greater than 5. After the timeout message is received, however, it will handle any message—including lower-priority messages that were received beforehand.

Translation on types

$$\lfloor \mathsf{ActorRef}(\langle \ell_i : A_i \rangle_i) \rfloor = \mathsf{ActorRef}(\langle \ell_i : \lfloor A_i \rfloor \rangle_i) \qquad \lfloor A \times B \rfloor = \lfloor A \rfloor \times \lfloor B \rfloor \qquad \lfloor A + B \rfloor = \lfloor A \rfloor + \lfloor B \rfloor$$

$$\lfloor \mu X.A \rfloor = \mu X.\lfloor A \rfloor \qquad \lfloor A \to^{C} B \rfloor = \lfloor A \rfloor \to^{\lfloor C \rfloor} \mathsf{List}(\lfloor C \rfloor) \to^{\lfloor C \rfloor} (\lfloor B \rfloor \times \mathsf{List}(\lfloor C \rfloor))$$

where $C = \langle \ell_i : A'_i \rangle_i$, and $\lfloor C \rfloor = \langle \ell_i : \lfloor A'_i \rfloor \rangle_i$

Translation on values

$$\lfloor \lambda x.M \rfloor = \lambda x.\lambda mb.(\lfloor M \rfloor\ mb) \qquad \lfloor \mathsf{rec}\, f(x)\,.\,M \rfloor = \mathsf{rec}\, f(x)\,.\,\lambda mb.(\lfloor M \rfloor\ mb)$$

Translation on computation terms (wrt. a mailbox type $\langle \ell_i : A_i \rangle_i$)

$$\begin{array}{rcl}
\lfloor V\ W \rfloor\ mb & = & \mathsf{let}\ f \Leftarrow (\lfloor V \rfloor\ \lfloor W \rfloor)\ \mathsf{in}\ f\ mb \\
\lfloor \mathsf{return}\ V \rfloor\ mb & = & \mathsf{return}\ (\lfloor V \rfloor, mb) \\
\lfloor \mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N \rfloor\ mb & = & \mathsf{let}\ resPair \Leftarrow \lfloor M \rfloor\ mb\ \mathsf{in}\ \mathsf{let}\ (x, mb') = resPair\ \mathsf{in}\ \lfloor N \rfloor\ mb' \\
\lfloor \mathsf{self} \rfloor\ mb & = & \mathsf{let}\ selfPid \Leftarrow \mathsf{self}\ \mathsf{in}\ \mathsf{return}\ (selfPid, mb) \\
\lfloor \mathsf{send}\ V\ W \rfloor\ mb & = & \mathsf{let}\ x \Leftarrow \mathsf{send}\ (\lfloor V \rfloor)\ (\lfloor W \rfloor)\ \mathsf{in}\ \mathsf{return}\ (x, mb) \\
\lfloor \mathsf{spawn}\ M \rfloor\ mb & = & \mathsf{let}\ spawnRes \Leftarrow \mathsf{spawn}(\lfloor M \rfloor\ \texttt{[ ]})\ \mathsf{in}\ \mathsf{return}\ (spawnRes, mb) \\
\lfloor \mathsf{receive}\ \{\overrightarrow{c}\} \rfloor\ mb & = & \texttt{find}(\overrightarrow{c}, mb)
\end{array}$$

Translation on configurations

$$\begin{array}{rcl}
\lfloor (\nu a)\mathcal{C} \rfloor & = & \{(\nu a)\mathcal{D} \mid \mathcal{D} \in \lfloor C \rfloor\} \\
\lfloor \mathcal{C}_1 \parallel \mathcal{C}_2 \rfloor & = & \{\mathcal{D}_1 \parallel \mathcal{D}_2 \mid \mathcal{D}_1 \in \lfloor \mathcal{C}_1 \rfloor \wedge \mathcal{D}_2 \in \lfloor \mathcal{C}_2 \rfloor\} \\
\lfloor \langle a, M, \overrightarrow{V} \rangle \rfloor & = & \{\langle a, \lfloor M \rfloor\ \texttt{[ ]}, \lfloor \overrightarrow{V} \rfloor \rangle\} \cup \\
 & & \{\langle a, (\lfloor M \rfloor\ \overrightarrow{W_i^1}), \overrightarrow{W_i^2} \rangle \mid i \in 1..n\}
\end{array}$$

where

$$\overrightarrow{W_i^1} = \lfloor V_1 \rfloor :: \ldots :: \lfloor V_i \rfloor :: \texttt{[ ]} \qquad \overrightarrow{W_i^2} = \lfloor V_{i+1} \rfloor \cdot \ldots \cdot \lfloor V_n \rfloor$$

Figure 17 Translation from λ_{act} with selective receive into λ_{act}.

Figure 16 shows the additional syntax, typing rule, and configuration reduction rule required to encode selective receive; the type Bool and logical operators are encoded using sums in the standard way. We write $\Gamma \vdash_{\mathsf{P}} M : A$ to mean that under context Γ, a term M which does not perform any communication or concurrency actions has type A. Intuitively, this means that no subterm of M is a communication or concurrency construct.

The receive $\{\overrightarrow{c}\}$ construct models an ordered sequence of receive pattern clauses c of the form $(\langle \ell = x \rangle\ \mathsf{when}\ M) \mapsto N$, which can be read as "If a message with body x has label ℓ and satisfies predicate M, then evaluate N". The typing rule for receive $\{\overrightarrow{c}\}$ ensures that for each pattern $\langle \ell_i = x_i \rangle\ \mathsf{when}\ M_i \mapsto N_i$ in $\overrightarrow{c}$, we have that there exists some $\ell_i : A_i$ contained in the mailbox variant type; and when Γ is extended with $x_i : A_i$, that the guard M_i has type Bool and the body N_i has the same type C for each branch.

The reduction rule for selective receive is inspired by that of Fredlund [16]. Assume that the mailbox is of the form $V_1 \cdot \ldots \cdot V_k \cdot \ldots V_n$, with $\overrightarrow{W} = V_1 \cdot \ldots \cdot V_{k-1}$ and $\overrightarrow{W'} = V_{k+1} \cdot \ldots \cdot V_n$. The $\mathsf{matches}(c, V)$ predicate holds if the label matches, and the branch guard evaluates to true. The $\mathsf{matchesAny}(\overrightarrow{c}, V)$ predicate holds if V matches any pattern in $\overrightarrow{c}$. The key idea is that V_k is the first value to satisfy a pattern. The construct evaluates to the body of the matched pattern, with the message payload V'_k substituted for the pattern variable x_k; the final mailbox is $\overrightarrow{W} \cdot \overrightarrow{W'}$ (that is, the original mailbox without V_k).

Reduction in the presence of selective receive preserves typing.

▶ **Theorem 30** (Preservation (λ_{act} configurations with selective receive)). *If* $\Gamma; \Delta \mid \langle \ell_i : A_i \rangle_i \vdash \mathcal{C}_1$ *and* $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$, *then* $\Gamma; \Delta \mid \langle \ell_i : A_i \rangle_i \vdash \mathcal{C}_2$.

Translation to λ_{act}. Given the additional constructs used to translate λ_{ch} into λ_{act}, it is possible to translate λ_{act} with selective receive into plain λ_{act}. Key to the translation is

$$
\begin{array}{l}
\texttt{find}(\overrightarrow{c}, mb) \triangleq \\
(\text{rec } \mathit{findLoop}(ms)\,. \\
\quad \text{let } (mb_1, mb_2) = ms \text{ in} \\
\quad \text{case } mb_2\ \{ \\
\qquad [\,] \mapsto \texttt{loop}(\overrightarrow{c}, mb_1) \\
\qquad x :: mb_2' \mapsto \\
\qquad\quad \text{let } mb' \Leftarrow mb_1 +\!\!+ mb_2' \text{ in} \\
\qquad\quad \text{case } x\ \{\texttt{branches}(\overrightarrow{c}, mb', \\
\qquad\qquad \lambda y.(\text{let } mb_1' \Leftarrow mb_1 +\!\!+ [y] \text{ in} \\
\qquad\qquad\quad \mathit{findLoop}\,(mb_1', mb_2'))))\})\,([\,], mb)
\end{array}
$$

$$
\begin{array}{l}
\texttt{ifPats}(mb, \ell, y, \epsilon, \mathit{default}) = \mathit{default}\,\langle \ell = y \rangle \\
\texttt{ifPats}(mb, \ell, y, \\
\quad (\langle \ell = x \rangle \text{ when } M \mapsto N) \cdot \mathit{pats}, \mathit{default}) = \\
\quad\quad \text{let } \mathit{resPair} \Leftarrow (\lfloor M \rfloor\, mb)\{y/x\} \text{ in} \\
\quad\quad \text{let } (\mathit{res}, mb') = \mathit{resPair} \text{ in} \\
\quad\quad \text{if } \mathit{res} \text{ then } (\lfloor N \rfloor\, mb)\{y/x\} \\
\quad\quad \text{else } \texttt{ifPats}(mb, \ell, y, \mathit{pats}, \mathit{default})
\end{array}
$$

$$
\begin{array}{l}
\texttt{loop}(\overrightarrow{c}, mb) \triangleq \\
\quad (\text{rec } \mathit{recvLoop}(mb)\,. \\
\quad\quad \text{let } x \Leftarrow \text{receive in} \\
\quad\quad \text{case } x\ \{\texttt{branches}(\overrightarrow{c}, mb, \\
\quad\quad\quad \lambda y.\text{let } mb' \Leftarrow mb +\!\!+ [y] \text{ in} \\
\quad\quad\quad\quad \mathit{recvLoop}\, mb')\})\, mb
\end{array}
$$

$$\texttt{label}(\langle \ell = x \rangle \text{ when } M \mapsto N) = \ell$$

$$\texttt{labels}(\overrightarrow{c}) = \texttt{noDups}([\texttt{label}(c) \mid c \leftarrow \overrightarrow{c}])$$

$$\texttt{matching}(\ell, \overrightarrow{c}) = [c \mid (c \leftarrow \overrightarrow{c}) \wedge \texttt{label}(c) = \ell]$$

$$\texttt{unhandled}(\overrightarrow{c}) = [\ell \mid (\langle \ell : A \rangle \leftarrow \langle \ell_i : A_i \rangle_i) \wedge \ell \notin \texttt{labels}(\overrightarrow{c})]$$

$$\texttt{branches}(\overrightarrow{c}, mb, \mathit{default}) = \texttt{patBranches}(\overrightarrow{c}, mb, \mathit{default}) \cdot \texttt{defaultBranches}(\overrightarrow{c}, mb, \mathit{default})$$

$$
\begin{array}{l}
\texttt{patBranches}(\overrightarrow{c}, mb, \mathit{default}) = \\
\quad [\langle \ell = x \rangle \mapsto \texttt{ifPats}(mb, \ell, x, \overrightarrow{c_\ell}, \mathit{default}) \mid (\ell \leftarrow \texttt{labels}(\overrightarrow{c}))\ \wedge \overrightarrow{c_\ell} = \texttt{matching}(\ell, \overrightarrow{c}) \wedge x \text{ fresh}]
\end{array}
$$

$$\texttt{defaultBranches}(\overrightarrow{c}, mb, \mathit{default}) = [\langle \ell = x \rangle \mapsto \mathit{default}\,\langle \ell = x \rangle \mid (\ell \leftarrow \texttt{unhandled}(\overrightarrow{c})) \wedge x \text{ fresh}]$$

Figure 18 Meta level definitions for translation from λ_{act} with selective receive to λ_{act} (wrt. a mailbox type $\langle \ell_i : A_i \rangle_i$).

reasoning about values in the mailbox at the term level; we maintain a term-level 'save queue' of values that have been received but not yet matched, and can loop through the list to find the first matching value. Our translation is similar in spirit to the "stashing" mechanism described by Haller [19] to emulate selective receive in Akka, where messages can be moved to an auxiliary queue for processing at a later time.

Figure 17 shows the translation formally. Except for function types, the translation on types is homomorphic. Similar to the translation from λ_{act} into λ_{ch}, we add an additional parameter for the save queue.

The translation on terms $\lfloor M \rfloor\, mb$ takes a variable mb representing the save queue as its parameter, returning a pair of the resulting term and the updated save queue. The majority of cases are standard, except for receive $\{\overrightarrow{c}\}$, which relies on the meta-level definition $\texttt{find}(\overrightarrow{c}, mb)$: $\overrightarrow{c}$ is a sequence of clauses, and mb is the save queue. The constituent *findLoop* function takes a pair of lists (mb_1, mb_2), where mb_1 is the list of processed values found not to match, and mb_2 is the list of values still to be processed. The loop inspects the list until one either matches, or the end of the list is reached. Should no values in the term-level representation of the mailbox match, then the `loop` function repeatedly receives from the mailbox, testing each new message against the patterns.

Note that the case construct in the core λ_{act} calculus is more restrictive than selective receive: given a variant $\langle \ell_i : A_i \rangle_i$, case requires a single branch for each label. Selective receive allows multiple branches for each label, each containing a possibly-different predicate, and does not require pattern matching to be exhaustive.

We therefore need to perform pattern matching elaboration; this is achieved by the `branches` meta level definition. We make use of list comprehension notation: for example,

$[c \mid (c \leftarrow \overrightarrow{c}) \wedge \texttt{label}(c) = \ell]$ returns the (ordered) list of clauses in a sequence $\overrightarrow{c}$ such that the label of the receive clause matches a label ℓ. We assume a meta level function **noDups** which removes duplicates from a list. Case branches are computed using the **branches** meta level definition: **patBranches** creates a branch for each label present in the selective receive, creating (via **ifPats**) a sequence of if-then-else statements to check each predicate in turn; **defaultBranches** creates a branch for each label that is present in the mailbox type but not in any selective receive clauses.

Properties of the translation. The translation preserves typing of terms and values.

▶ **Lemma 31** (Translation preserves typing (values and terms)).

1. *If* $\Gamma \vdash V : A$, *then* $\lfloor\Gamma\rfloor \vdash \lfloor V\rfloor : \lfloor A\rfloor$.
2. *If* $\Gamma \mid \langle \ell_i : A_i \rangle_i \vdash M : B$, *then* $\lfloor\Gamma\rfloor, mb : \mathsf{List}(\langle \ell_i : \lfloor A_i\rfloor\rangle_i) \mid \langle \ell_i : \lfloor A_i\rfloor\rangle_i \vdash \lfloor M\rfloor mb : (\lfloor B\rfloor \times \mathsf{List}(\langle \ell_i : \lfloor A_i\rfloor\rangle_i))$.

Alas, a direct one-to-one translation on configurations is not possible, since a message in a mailbox in the source language could be either in the mailbox or the save queue in the target language. Consequently, we translate a configuration into a set of possible configurations, depending on how many messages have been processed. We can show that all configurations in the resulting set are type-correct, and can simulate the original reduction.

▶ **Theorem 32** (Translation preserves typing). *If* $\Gamma; \Delta \vdash \mathcal{C}$, *then* $\forall \mathcal{D} \in \lfloor\mathcal{C}\rfloor$, *it is the case that* $\lfloor\Gamma\rfloor; \lfloor\Delta\rfloor \vdash \mathcal{D}$.

▶ **Theorem 33** (Simulation (λ_{act} with selective receive in λ_{act})). *If* $\Gamma; \Delta \vdash \mathcal{C}$ *and* $\mathcal{C} \longrightarrow \mathcal{C}'$, *then* $\forall \mathcal{D} \in \lfloor\mathcal{C}\rfloor$, *there exists a* $\mathcal{D}'$ *such that* $\mathcal{D} \longrightarrow^{+} \mathcal{D}'$ *and* $\mathcal{D}' \in \lfloor\mathcal{C}'\rfloor$.

Remark. Originally we expected to need to add an analogous selective receive construct to λ_{ch} in order to be able to translate λ_{act} with selective receive into λ_{ch}. We were surprised (in part due to the complex reduction rule and the native runtime support in Erlang) when we discovered that selective receive can be emulated in plain λ_{act}. Moreover, we were pleasantly surprised that types pose no difficulties in the translation.

7.3 Choice

The calculus λ_{ch} supports only blocking receive on a *single* channel. A more powerful mechanism is *selective communication*, where a value is taken nondeterministically from *two* channels. An important use case is receiving a value when either channel could be empty.

Here we have considered only the most basic form of selective choice over two channels. More generally, it may be extended to arbitrary regular data types [42]. As Concurrent ML [45] embraces rendezvous-based synchronous communication, it provides *generalised selective communication* where a process can synchronise on a mixture of input or output communication events. Similarly, the join patterns of the join calculus [14] provide a general abstraction for selective communication over multiple channels.

As we are working in the asynchronous setting where a give operation can reduce immediately, we consider only input-guarded choice. Input-guarded choice can be added straightforwardly to λ_{ch}, as shown in Figure 19. Emulating such a construct satisfactorily in λ_{act} is nontrivial, because messages must be multiplexed through a local queue. One approach could be to use the work of Chaudhuri [8] which shows how to implement generalised choice using synchronous message passing, but implementing this in λ_{ch} may be difficult due to the asynchrony of give. We leave a more thorough investigation to future work.

$$\frac{\Gamma \vdash V : \mathsf{ChanRef}(A) \quad \Gamma \vdash W : \mathsf{ChanRef}(B)}{\Gamma \vdash \mathsf{choose}\, V\, W : A + B}$$

$$\begin{array}{rcl} E[\mathsf{choose}\, a\, b] \parallel a(W_1 \cdot \overrightarrow{V_1}) \parallel b(\overrightarrow{V_2}) & \longrightarrow & E[\mathsf{return}\, (\mathsf{inl}\, W_1)] \parallel a(\overrightarrow{V_1}) \parallel b(\overrightarrow{V_2}) \\ E[\mathsf{choose}\, a\, b] \parallel a(\overrightarrow{V_1}) \parallel b(W_2 \cdot \overrightarrow{V_2}) & \longrightarrow & E[\mathsf{return}\, (\mathsf{inr}\, W_2)] \parallel a(\overrightarrow{V_1}) \parallel b(\overrightarrow{V_2}) \end{array}$$

Figure 19 Additional typing and evaluation rules for λ_{ch} with choice.

7.4 Behavioural types

Behavioural types allow the type of an object (e.g. a channel) to evolve as a program executes. A widely studied behavioural typing discipline is that of *session types* [26, 27], which are channel types sufficiently expressive to describe *communication protocols* between participants. For example, the session type for a channel which sends two integers and receives their sum could be defined as !Int.!Int.?Int.end. Session types are suited to channels, whereas current work on session-typed actors concentrates on runtime monitoring [39].

A natural question to ask is whether one can combine the benefits of actors and of session types—indeed, this was one of our original motivations for wanting to better understand the relationship between actors and channels in the first place! A session-typed channel may support both sending and receiving (at different points in the protocol it encodes), but communication with another process' mailbox is one-way. We have studied several variants of λ_{act} with *polarised* session types [43, 36] which capture such one-way communication, but they seem too weak to simulate session-typed channels. In future, we would like to find an extension of λ_{act} with behavioural types that admits a similar simulation result to the ones in this paper.

8 Related work

Our formulation of concurrent λ-calculi is inspired by $\lambda(\mathsf{fut})$ [40], a concurrent λ-calculus with threads, futures, reference cells, and an atomic exchange construct. In the presence of lists, futures are sufficient to encode asynchronous channels. In λ_{ch}, we concentrate on asynchronous channels to better understand the correspondence with actors. Channel-based concurrent λ-calculi form the basis of functional languages with session types [17, 35].

Concurrent ML [45] extends Standard ML with a rich set of combinators for synchronous channels, which again can emulate asynchronous channels. A core notion in Concurrent ML is nondeterministically synchronising on multiple synchronous events, such as sending or receiving messages; relating such a construct to an actor calculus is nontrivial, and remains an open problem. Hopac [28] is a channel-based concurrency library for F#, based on Concurrent ML. The Hopac documentation relates synchronous channels and actors [1], implementing actor-style primitives using channels, and channel-style primitives using actors. The implementation of channels using actors uses mutable references to emulate the take function, whereas our translation achieves this using message passing. Additionally, our translation is formalised and we prove that the translations are type- and semantics-preserving.

Links [9] provides actor-style concurrency, and the paper describes a translation into $\lambda(\mathsf{fut})$. Our translation is semantics-preserving and can be done without synchronisation.

The actor model was designed by Hewitt [23] and examined in the context of distributed systems by Agha [2]. Agha et al. [3] describe a functional actor calculus based on the λ-calculus augmented by three core constructs: send sends a message; letactor creates a new

actor; and become changes an actor's behaviour. The operational semantics is defined in terms of a global actor mapping, a global multiset of messages, a set of *receptionists* (actors which are externally visible to other configurations), and a set of external actor names. Instead of become, we use an explicit receive construct, which more closely resembles Erlang (referred to by the authors as "essentially an actor language"). Our concurrent semantics, more in the spirit of process calculi, encodes visibility via name restrictions and structural congruences. The authors consider a behavioural theory in terms of operational and testing equivalences—something we have not investigated.

Scala has native support for actor-style concurrency, implemented efficiently without explicit virtual machine support [20]. The actor model inspires *active objects* [33]: objects supporting asynchronous method calls which return responses using futures. De Boer et al. [10] describe a language for active objects with cooperatively scheduled threads within each object. Core ABS [32] is a specification language based on active objects. Using futures for synchronisation sidesteps the type pollution problem inherent in call-response patterns with actors, although our translations work in the absence of synchronisation. By working in the functional setting, we obtain more compact calculi.

9 Conclusion

Inspired by languages such as Go which take channels as core constructs for communication, and languages such as Erlang which are based on the actor model of concurrency, we have presented translations back and forth between a concurrent λ-calculus λ_{ch} with channel-based communication constructs and a concurrent λ-calculus λ_{act} with actor-based communication constructs. We have proved that λ_{act} can simulate λ_{ch} and vice-versa.

The translation from λ_{act} to λ_{ch} is straightforward, whereas the translation from λ_{ch} to λ_{act} requires considerably more effort. Returning to Figure 2, this is unsurprising!

We have also shown how to extend λ_{act} with synchronisation, greatly simplifying the translation from λ_{ch} into λ_{act}, and have shown how Erlang-style selective receive can be emulated in λ_{act}. Additionally, we have discussed input-guarded choice in λ_{ch}, and how behavioural types may fit in with λ_{act}.

In future, we firstly plan to strengthen our operational correspondence results by considering operational completeness. Secondly, we plan to investigate how to emulate λ_{ch} with input-guarded choice in λ_{act}. Finally, we intend to use the lessons learnt from studying λ_{ch} and λ_{act} to inform the design of an actor-inspired language with behavioural types.

Acknowledgements. Thanks to Philipp Haller, Daniel Hillerström, Ian Stark, and the anonymous reviewers for detailed comments.

References

1 Actors and Hopac, 2016. URL: `https://www.github.com/Hopac/Hopac/blob/master/Docs/Actors.md`.

2 Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

3 Gul A Agha, Ian A Mason, Scott F Smith, and Carolyn L Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(01):1–72, 1997.

4 Akka Typed, 2016. URL: `http://doc.akka.io/docs/akka/current/scala/typed.html`.

5 Elvira Albert, Puri Arenas, and Miguel Gómez-Zamalloa. Testing of concurrent and imperative software using clp. In *PPDP*, pages 1–8. ACM, 2016.

6 Joe Armstrong. *Making reliable distributed systems in the presence of sodware errors*. PhD thesis, The Royal Institute of Technology Stockholm, Sweden, 2003.

7 Francesco Cesarini and Steve Vinoski. *Designing for Scalability with Erlang/OTP*. " O'Reilly Media, Inc.", 2016.

8 Avik Chaudhuri. A Concurrent ML Library in Concurrent Haskell. In *ICFP*, pages 269–280, New York, NY, USA, 2009. ACM.

9 Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web Programming Without Tiers. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *FMCO*, volume 4709, pages 266–296. Springer Berlin Heidelberg, 2007.

10 Frank S De Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In *ESOP*, pages 316–330. Springer, 2007.

11 Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 43 Years of Actors: A Taxonomy of Actor Models and Their Key Properties. In *AGERE*. ACM, 2016.

12 Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(1-3):35–75, 1991.

13 Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI*, pages 237–247. ACM, 1993.

14 Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *POPL*, pages 372–385. ACM Press, 1996. URL: `http://dl.acm.org/citation.cfm?id=237721`, `doi:10.1145/237721.237805`.

15 Simon Fowler, Sam Lindley, and Philip Wadler. Mixing Metaphors: Actors as Channels and Channels as Actors (Extended Version). *CoRR*, abs/1611.06276, 2017. URL: `http://arxiv.org/abs/1611.06276`.

16 Lars-Åke Fredlund. *A framework for reasoning about Erlang code*. PhD thesis, The Royal Institute of Technology Stockholm, Sweden, 2001.

17 Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20:19–50, January 2010.

18 David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *LFP*, pages 28–38. ACM, 1986.

19 Philipp Haller. On the integration of the actor model in mainstream technologies: the Scala perspective. In *AGERE*, pages 1–6. ACM, 2012.

20 Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2):202–220, 2009.

21 Paul Harvey. *A linguistic approach to concurrent, distributed, and adaptive programming across heterogeneous platforms*. PhD thesis, University of Glasgow, 2015.

22 Jiansen He, Philip Wadler, and Philip Trinder. Typecasting actors: From Akka to TAkka. In *SCALA*, pages 23–33. ACM, 2014.

23 Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

24 Rich Hickey. Clojure core.async Channels, 2013. URL: `http://clojure.com/blog/2013/06/28/clojure-core-async-channels.html`.

25 C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

26 Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer Berlin Heidelberg, 1993.

27 Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *ESOP*, chapter 9, pages 122–138. Springer Berlin Heidelberg, Berlin/Heidelberg, 1998.

28 Hopac, 2016. URL: http://www.github.com/Hopac/hopac.

29 How are Akka actors different from Go channels?, 2013. URL: https://www.quora.com/How-are-Akka-actors-different-from-Go-channels.

30 Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in java. In *ECOOP*, pages 516–541. Springer, 2008.

31 Is Scala's actors similar to Go's coroutines?, 2014. URL: http://stackoverflow.com/questions/22621514/is-scalas-actors-similar-to-gos-coroutines.

32 Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In *FMCO*, pages 142–164. Springer, 2010.

33 R. Greg Lavender and Douglas C. Schmidt. Active object: An object behavioral pattern for concurrent programming. In John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors, *Pattern Languages of Program Design 2*, pages 483–499. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996. URL: http://dl.acm.org/citation.cfm?id=231958.232967.

34 Paul B. Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Information and Computation*, 185(2):182–210, 2003.

35 Sam Lindley and J. Garrett Morris. A Semantics for Propositions as Sessions. In *ESOP*, pages 560–584. Springer, 2015.

36 Sam Lindley and J. Garrett Morris. Embedding session types in haskell. In *Haskell*, pages 133–145. ACM, 2016.

37 Massimo Merro and Davide Sangiorgi. On asynchrony in name-passing calculi. *Mathematical Structures in Computer Science*, 14(5):715–767, 2004. doi:10.1017/S0960129504004323.

38 Robin Milner. *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press, 1st edition, June 1999.

39 Rumyana Neykova and Nobuko Yoshida. Multiparty session actors. In *COORDINATION*, pages 131–146. Springer, 2014.

40 Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364(3):338–356, 2006.

41 Luca Padovani and Luca Novara. Types for Deadlock-Free Higher-Order Programs. In Susanne Graf and Mahesh Viswanathan, editors, *FORTE*, pages 3–18. Springer International Publishing, 2015.

42 Jennifer Paykin, Antal Spector-Zabusky, and Kenneth Foner. `choose` your own derivative. In *TyDe*, pages 58–59. ACM, 2016.

43 Frank Pfenning and Dennis Griffith. Polarized substructural session types. In *FoSSaCS*, volume 9034 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2015.

44 Proto.Actor, 2016. URL: http://www.proto.actor.

45 John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 2007.

46 Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2003.

47 Typed Actors, 2016. URL: https://github.com/knutwalker/typed-actors.

48 Philip Wadler. Propositions as sessions. *Journal of Functional Programming*, 24(2-3):384–418, 2014.